POLITECNICO DI TORINO

Tesi di Laurea Magistrale

DAUIN
Corso di Laurea Magistrale in Mechatronic Engineering

Fast Adversarial Training for Deep Neural Networks

Nikfam Farzad

Relatore: **Prof. Martina Maurizio**
Correlatore: **Prof. Shafique Muhammad**
Tutore: **Dott. Marchisio Alberto**

Marzo 2020

Index

Abstract VII

Part I - General Introduction 1

1 – Machine learning 3

1.1 What is machine learning 3
1.2 Unsupervised learning 3
1.3 Supervised learning 4
1.3.1 Regression 4
1.3.2 Classification 5
1.4 Cost function 6
1.5 Gradient descent 6
1.5.1 Batch gradient descent (BGD) 7
1.5.2 Stochastic gradient descent (SGD) 7
1.6 Normal equation 8
1.7 Hyperparameters 9
1.7.1 Learning rate 9
1.7.2 Momentum 10
1.7.3 Batch size 10
1.7.4 Weight decay 10
1.7.5 Epochs 11
1.8 Datasets 11
1.8.1 MNIST 12
1.8.2 CIFAR10 12
1.8.3 CIFAR100 13
1.8.4 ImageNet 13
1.9 Linear regression 13
1.10 Logistic regression 15
1.11 Neural network 16
1.11.1 Convolutional neural network (CNN) 18
1.12 Problems and resolutions 19

1.12.1 Features scaling 19
1.12.2 Mean normalization 20
1.12.3 Learning rate problems 20
1.12.3.1 Learning rate finder 21
1.12.4 Training problems 21
1.12.4.1 Underfitting 23
1.12.4.2 Overfitting 23
1.12.5 Random initialization 24

2 – Python & TensorFlow 25

2.1 Programming languages 25
2.2 MATLAB 25
2.3 Python 26
2.3.1 Self parameter 27
2.4 TensorFlow 28
2.4.1 TensorFlow functions 28
2.4.1.1 Piecewise constant 29
2.4.1.2 Exponential decay 29
2.4.1.3 Polynomial decay 30
2.4.2 TensorBoard 31
2.5 PyTorch 31
2.6 Keras 32

Part II - Research Field 33

3 – Adversarial training 35

3.1 Why do we need adversarial training? 35
3.2 Adversarial examples 35
3.3 Adversarial attacks 36
3.3.1 White-box attacks 38
3.3.2 Black-box attacks 39
3.4 Training against adversarial attacks 39
3.4.1 Data augmentation 39
3.4.2 Defensive distillation 39
3.4.3 Second model control 39
3.5 Adversarial libraries 40
3.6 Free Adversarial Training (FAT) 41

4 – Fast training 45

4.1 Why fast? 45
4.1.1 Fast training techniques 45
4.2 1 Cycle policy 46
4.2.1 1 Cycle – Learning rate 46
4.2.2 1 Cycle – Momentum 47
4.2.3 1 Cycle – Other hyperparameters 48

4.3 Cyclical policy 49
4.3.1 Cycle length 49
4.3.2 Cycle boundary values 50
4.4 Warm restarts 51
4.5 Other implementations 52
5 – Fast adversarial training 53
5.1 Super model 53
5.2 FAT results 53
5.3 Hyperparameters tested 55
5.3.1 Learning rate's shapes summary 56
5.3.2 Momentum's shapes summary 57
5.4 Simulations 57
5.4.1 Natural images results 59
5.4.2 Adversarial images results 62
5.5 Conclusion 63
5.5.1 Future works 63
Bibliography 65

Abstract

Thesis topic focuses on Machine Learning from the software point of view, nowadays one of the research route for the management of large databases. Machine Learning is already widely present in our daily lives and we can find it, for example, both in the anti-spam filter of electronic mail, and in facial recognition of cameras, in the automatic corrector of smartphones, or in weather forecasts, etc.

The aim of the thesis is to review algorithms written in Python language for models robust to adversarial attacks and try to apply to them fast training techniques to improve computational time.
The word “fast training” refers to a code able to reach the skill to distinguish and divide a large database's data in a reasonable time according to the learning rules given by the programmer. The main criticality of fast training consists in being able to find a quite fast algorithm but as well accurate: too much accuracy may require learning times that are too long to be acceptable, while a high convergence speed could lead to wrong results or even worse, do not converge, but diverge.
Instead “adversarial training” means a code that can be robust against data modified on purpose that can not be distinguished by a human, but can have very negative effects on a Deep Neural Network (DNN) model, for example an attack can modify some pixels of an image without any real difference for a human eye but completely misclassified by the DNN model.

The first steps were to study the basic concepts of the Machine Learning, then going on to compose simple codes in Matlab, where the fluidity allows a faster learning, to finally review and write more complex algorithms in Python, which language flexibility allows various options on them.
The main libraries used in python for Machine Learning are TensorFlow (provided by Google) and Pytorch (provided by Facebook). In this thesis there is a focus on TensorFlow that allows to concentrate on the problem using a very high level language that optimizes the computational effort of the machine.
The principal model used for this purpose is the DNN, that is a virtual representation and simplification of human brain. This model works cyclically:

this means that it is trained on a Training dataset of images and then tested on a Validation dataset. Due to computational limits and to the huge dimensions of the datasets, it can take from a few hours to several days to perform a training session, therefore the goal of this thesis is to speed up the training time.

The general method to improve the performances of the training is the fine tuning of hyperparameters, like as: learning rate, momentum, weight decay, batch size, etc. Learning rate is the most important hyperparameter and by modifying its shape and value during the training we can obtain a robust model to adversarial attack up to 2 times faster than normal training. The tests were performed on 2 different datasets (CIFAR10, CIFAR100) and with various shapes to obtain the best results with a “trial and error” approach.

Part I

-

General Introduction

1 – Machine learning

1.1 What is machine learning

Machine learning [1] [2] [3] is a branch of artificial intelligence present in our daily life and which increasingly integrates with the technological systems we use nowadays. The use of machine learning can be seen in weather forecast, in email spam filter, in cameras facial recognition, in smartphones automatic corrector, etc.
Machine learning is based on mathematical algorithms implemented through software that allow a code to self-adapt to data allowing the subdivision or recognition of the aforementioned data.
The main purpose of machine learning is to manage the large quantities of data that are produced by large companies every day, in order to manage information faster and efficiently.
Machine learning can be applied to the most varied types of data. During this thesis we will mainly consider images, therefore codes able to recognize and classify images.
Machine learning is divided into two main sections:

- Unsupervised learning
- Supervised learning

1.2 Unsupervised learning

We talk about unsupervised learning (Figure 1) when a model is not previously trained on a dataset and therefore tries to automatically recognize and divide new data that it has never seen before. Obviously, the model will already know the type of data it will receive, but the classification of the latter depends on the model itself. For example, you can use this approach when you want to divide the sound of an audio recording from the background noise or you want to divide the various voices present. In general it is not a widely used method, as it is less effective than

supervised learning, but in cases where there isn't previous data it can help in the classification of new input.

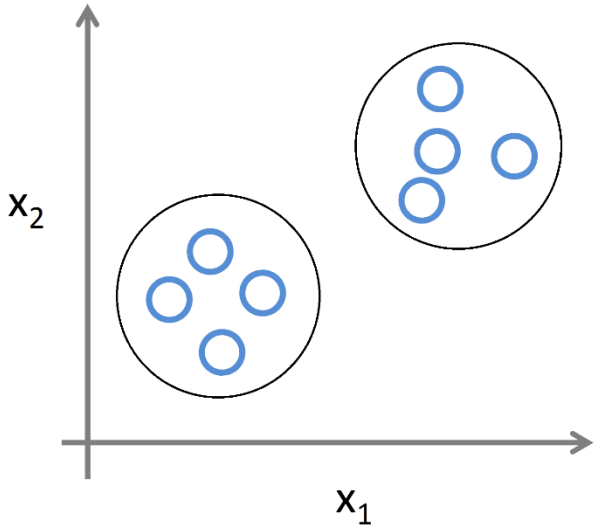

Figure 1 – Unsupervised learning division

1.3 Supervised learning

The term supervised learning (Figure 2) refers to the machine learning model mainly used, that is, based on the previous history of the data. In practice, in the case of supervised learning, a code is trained on an already existing dataset collected in a database, every single data item of the aforementioned database is correctly labelled. When the code is trained all labels are shown correct, at the end of the training the code is tested on a new dataset and its accuracy is calculated with the number of labels it can correctly predict.

Supervised learning can generate two types of results:

- Regression
- Classification

1.3.1 Regression

In the case of regression supervised learning the system output will be continuous, for example in the case of weather forecasts between the extreme cases of "good weather" and "bad weather" there can be all values intermediate where the

weather is partially cloudy with more or less sun. Regression is the trend of a curve that is plotted based on previous data, in order to predict the future. For example, if you analyze house prices on the basis of their size, you can make a price forecast for a new house whose size is known and you want to obtain an economic value in order to sell it. In carrying out this thesis we will focus mainly on the classification case.

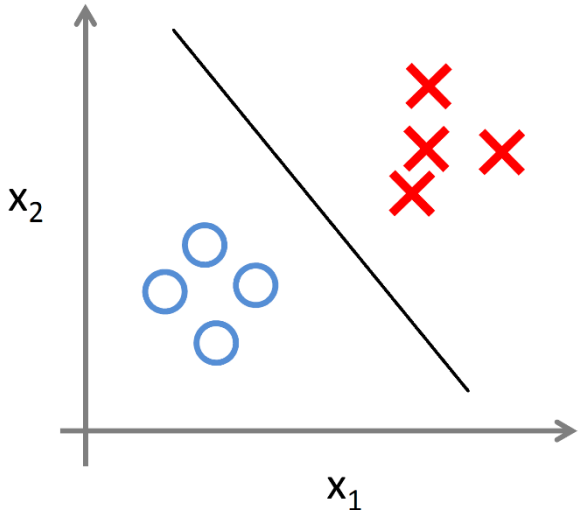

Figure 2 – Supervised learning classification

1.3.2 Classification

In a model created for classification, as the term itself says, the goal is to divide the data into classes. The most classic case is the binary subdivision: 0 and 1 that is to attribute to each data the label True or False. An example applicable to healthcare is the classification of tumors into benign and malignant. Using a supervised machine learning model for the classification you can show CAT images to the model, which will produce an output with 0 or 1 to indicate benignity or malignancy of the tumor. In the event that it is necessary to classify a greater number of data, such as in the case of the MNIST database, which contains 60 thousand images of handwritten digits from 0 to 9, it is possible to perform a mathematical trick to associate an intermediate value between 0 and 1 for each label and the one with the highest value becomes the label to be associated with that specific data.

1.4 Cost function

The cost function (Figure 3) is the equation that defines the error of the prediction from the correct label, that is actually how far it is from the minimum point. The cost function can be defined with different mathematical formulas, it depends on the type of algorithm you want to use. The purpose of machine learning is to try to minimize the cost function, that is, the more the error approaches zero, the more the model can correctly predict the data to be classified. If the cost function is three-dimensional, it means that you have to reach the valleys, the so-called minimum, to have the least possible error. If the cost function is not as simple as a hyperbolic paraboloid then the function will have several local minimum points and only one global minimum point. Often the difficulty is to build models capable of "jumping" out of a local minimum to try to go towards a deeper minimum. When training is performed on large datasets, the training trend is asymptotic: 100% accuracy can only be achieved indefinitely, therefore training is often interrupted after a number of cycles that are considered reasonable to achieve a acceptable accuracy.

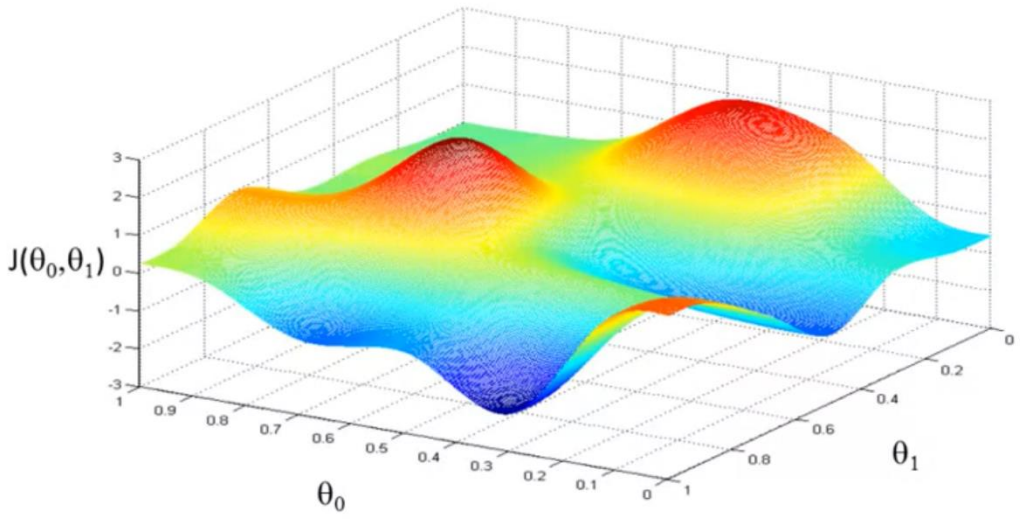

Figure 3 – 3-dimensional cost function

1.5 Gradient descent

Gradient descent is one of the main techniques that allows you to find the minimum of the cost function. Its role is to proceed with each iteration towards an area with a negative slope, therefore it exploits the derivative that allows each time

to find the angular coefficient of the function for each point. If the cost function has a number of variables higher than 2 then instead of the derivative the partial derivatives are calculated in order to obtain the gradient, hence the name gradient descent. Here is a typical gradient descent equation:

$$\theta_J = \theta_J - \alpha \cdot \frac{\partial}{\partial \theta_J} J(\theta)$$

The parameter "α" is the learning rate, that is the speed of the gradient descent, but we'll talk more about it in Chapter 1.7.1.

1.5.1 Batch gradient descent (BGD)

BGD (Figure 4) is mainly used for small datasets because at each step all the data are used to calculate the gradient descent and therefore slows down the training a lot. In the case of large datasets, mini-batch gradient descents are used, that is, a BGD calculated on a smallest portion of the dataset.

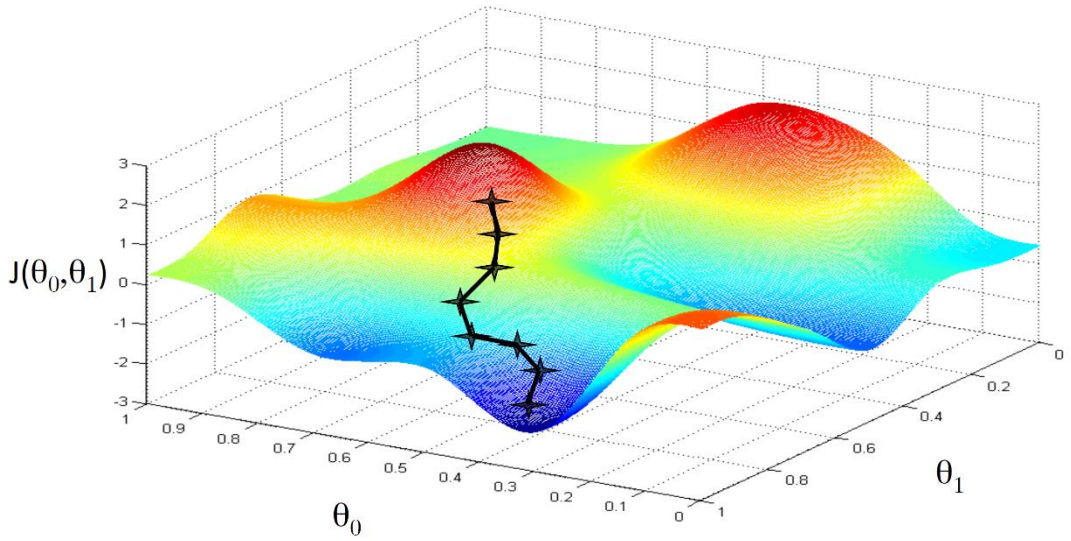

Figure 4 – Batch gradient descent

1.5.2 Stochastic gradient descent (SGD)

The SGD (Figure 5) function is to randomly search for an optimal path towards the minimum zone, this allows you to avoid calculating the gradient descent for

the whole database at each iteration, since it is a computationally expensive operation. So it is a technique used mainly for large datasets that does not clearly follow the path of descent to the shortest minimum, but takes longer to find the minimum, this is compensated by the fact that the iteration number to produce the SGD is much lower than a normal gradient descent.

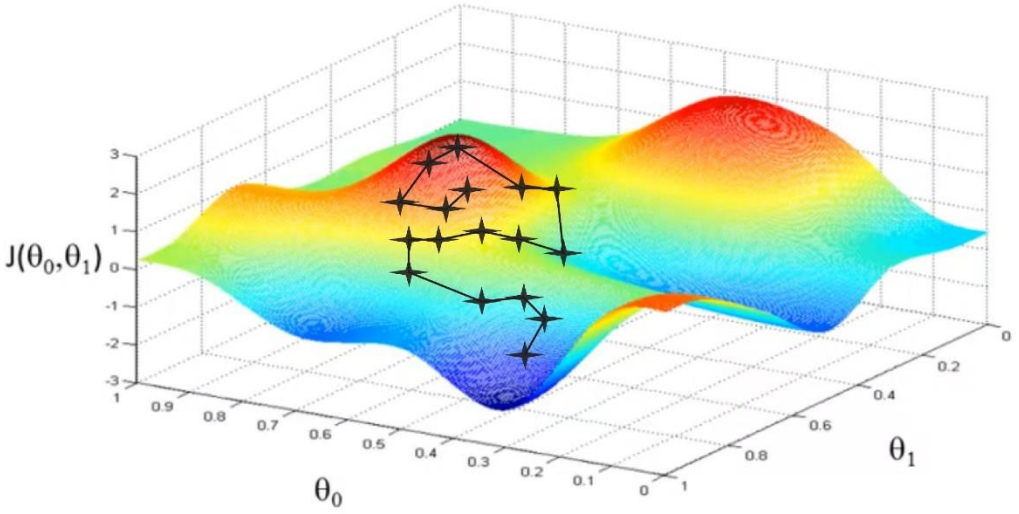

Figure 5 – Stochastic gradient descent

1.6 Normal equation

Normal equation is an alternative to the gradient descent that does not use the learning rate hyperparameter, does not need iterations, but simply uses matrices, as large as the features, that is, the system inputs. This method, however, is not often used because, despite it is more immediate for small datasets, it is not at all manageable on large amounts of data. In particular it is not easily adjustable and although it does not need many cycles, it still needs to calculate the inverse of some matrices, an algebraic operation often not recommended that can lead to singularities.

$$\theta = (x^T \cdot x)^{-1} \cdot x^T y$$

1.7 Hyperparameters

With hyperparameters [4] we mean all those parameters that directly influence a training, during the course of this thesis we will mainly consider the following hyperparameters:

- Learning rate
- Momentum
- Batch size
- Weight decay
- Epochs

1.7.1 Learning rate

The learning rate "α" (Figure 6) is the most important hyperparameter, since it controls the speed of the gradient descent in the minimum of the cost function. It appears in the gradient descent formula regardless of the cost function and its derivative, it is the angular coefficient of descent and can be varied during the training. The higher the value of the learning rate, the faster the gradient descends along the slopes, but if the value is too high, there is a risk of not going down to the minimum and diverging. To find the maximum value of the learning rate that can be used for a certain model and a given dataset, it is advisable to launch a test as a first training where the learning rate varies exponentially from very small values to very large values going through various orders of magnitude in order to find the optimal order of magnitude. Further on we will show you how to find the correct learning rate. In most trainings, the value of the learning rate starts to drop towards the end of the training in order to go deeper into the local minimum in which you are.

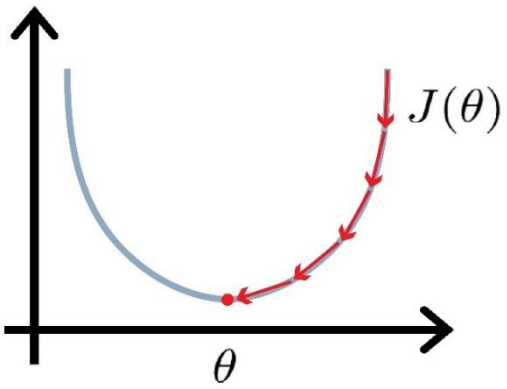

Figure 6 – Normal learning rate

1.7.2 Momentum

The momentum is a hyperparameter that is added to control other parameters during the training, therefore their weight in speeding up the training in order to quickly reach convergence. For example, the weight of the gradient descent can vary during the various iterations. It is not necessary to have momentum during a training, but it allows you to quickly check the weight of the equations within the code throughout the training period without having to intervene directly on the code for changes.

1.7.3 Batch size

The batch size [5] is a parameter used especially in large datasets because often the computational capacity of the computers does not allow to manage all the available data at the same time and therefore the training is iterated slowly over a dataset at a time, this set is the batch size. For example, if the dataset contains 50000 images and the batch size is set to 100, it means that for every single cycle 500 iterations will be performed, each with 100 different images of the dataset, so that for each cycle all the images are analyzed once each, but in blocks of 100 at a time. Often the batch size is set to a power of 2 because the calculators have an available memory calculated as a multiple of 2, obviously the larger the batch size the better the training, because the number of images compared at each iteration will be greater, but at the same time the training will become increasingly slow. A value for the batch size between 100 and 1000 is often more than acceptable as a compromise between total training time and quality of the result. The batch is one of the factors that directly affects the duration of the training.

1.7.4 Weight decay

The weight decay represents the speed with which the weight of some variables decays during the training process, specifically the weight of the coefficients of the variables is decided which are gradually less important than the constant term of the cost function. This weight loss is due to the fact that not all variables have the same importance within the cost function and some are more influential than others. A trivial, but easy-to-understand example can be: to determine if a patient has a certain type of tumor or not, all the input data are analyzed, for example: age, weight, gender, previous diseases, alcohol dependence / smoke, etc. , but not all these variables have the same importance for the tumor examined, certainly the hair color is almost not influential at all and therefore it is not even considered for evaluation purposes. This simple example is an application of the weight decay

based on the variables involved. During this thesis the weight decay has been kept fixed for all the training so as not to influence the other hyperparameters, but nothing prevents it from being modified during the training to try to obtain better results.

1.7.5 Epochs

Epochs [5] are probably the easiest hyperparameter to understand, i.e. they represent the total number of training cycles. During each epochs all data are analyzed at least once and all processes are performed. Each epoch is identical to the previous one, unless other hyperparameters are configured in such a way as to vary gradually with the passing of the epochs. Theoretically, if all the other hyperparameters are set correctly, then for an infinite number of epochs an accuracy of 100% on the data can be achieved, i.e. the learning curve is asymptotic and tends to infinity. Epochs, together with batch size, are one of the factors that directly influence the duration of the training. The training always lasts a finite and whole number of epochs and this is the equation for calculating cycles in 1 epoch:

$$1\ epoch = \frac{dataset\ size}{batch\ size}$$

1.8 Datasets

To work in the field of machine learning, it is necessary to have databases on which to perform the training. There are several ready to use, the main ones used during this thesis are:

- CIFAR10
- CIFAR100
- MNIST
- ImageNet

Each dataset is divided into 2 different sets:

- Training set: used to train the model in image recognition and to test its accuracy.
- Validation / test set: used as an external dataset to test the model on images never seen and therefore to verify that the model has not "become accustomed" only to the training set, but that it can also work on unknown data. Generally the accuracy is less than that obtained with the training set.

The number of data in a test set is generally equal to or less than the training set, so that much of the data is used for training.

1.8.1 MNIST

The MNIST [6] is a dataset consisting of 60 thousand images, divided into:

- 50 thousand images for training
- 10 thousand images for validation / test

The images are divided equally into 10 classes depicting the digits 0 to 9 written by hand (Figure 7) and therefore the goal of this dataset is to train models capable of recognizing and interpreting handwritten numbers. The images have a 28x28 pixels format in black and white. Part of this dataset was mainly used in the first part of this thesis.

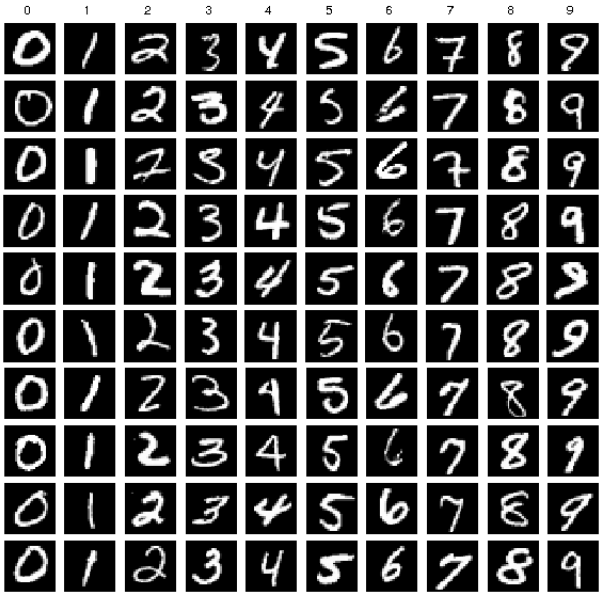

Figure 7 – An example of the MNIST dataset *[6]*

1.8.2 CIFAR10

The CIFAR10 [7] is also similar to the MNIST, but instead of using classes depicting handwritten digits, it has 10 classes depicting various things, including: airplanes, cars, dogs, cats, etc. (Figure 8) Each image is exclusive to the others, that is, a single image cannot represent two or more classes simultaneously. The size of each image is 32x32 pixels in color.

1.8.3 CIFAR100

The CIFAR100 [7] is a dataset similar to the CIFAR10 composed of 60 thousand images, equally distributed to the cifar10, but instead of 10 classes there are 100 different classes and therefore consequently it is more complicated to be able to obtain a good result during the training.

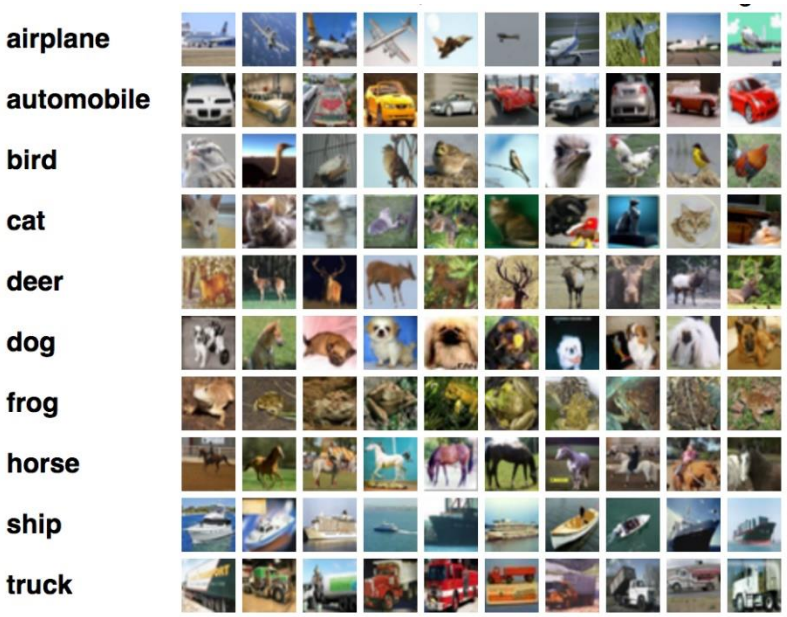

Figure 8 – An example of the CIFAR10 dataset *[7]*

1.8.4 ImageNet

It is one of the largest existing datasets [8] [9], it contains more than 1 million images, with various formats and divided into 1000 classes. This dataset is often used for the final validation of a model, but requires considerable hardware resources. During this thesis it has often been considered as a reference dataset, but never really used for the computational limits due to the computer in use.

1.9 Linear regression

Linear regression (Figure 9) is one of the simplest methods for approaching the world of machine learning and consists in dividing data, that is, drawing a line that can identify the delimitation area between the various groups.

It falls into the category of supervised regression learning and its constitutive function can always be simplified to a polynomial sum, where all the various inputs appear. The simplest form is:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

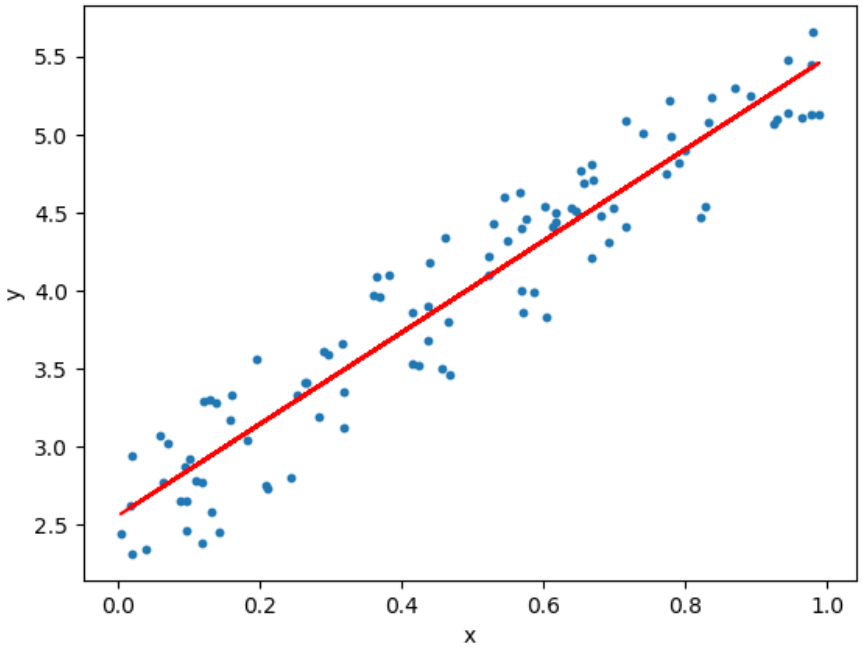

Figure 9 – Linear regression

The cost function in this case consist in the statistical variance, that is, an average of the square of the distance between the expected value (y) and the value obtained from training (h).

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

The gradient descent is obtained simply by updating the past gradient from time to time with the new gradient of the cost function damped with the learning rate hyperparameter.

$$\theta_J = \theta_J - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_J^{(i)} \quad with \quad x_0^{(i)} = 1$$

1.10 Logistic regression

Logistic regression falls into the category of supervised learning, but unlike the name, it is a model for classification. The term logistic refers to the type of function that allows you to switch from the continuous domain to the discrete one in order to obtain the classification, this function is also called Sigmoid (Figure 10).

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

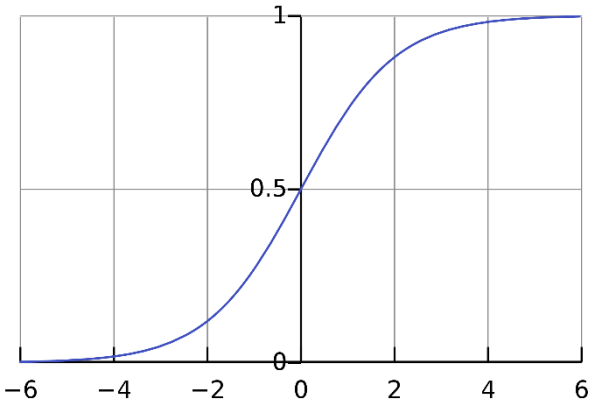

Figure 10 – Sigmoid function

As we can see, the sigmoid function classifies all inputs as output 0 or 1, rounding to the nearest integer.

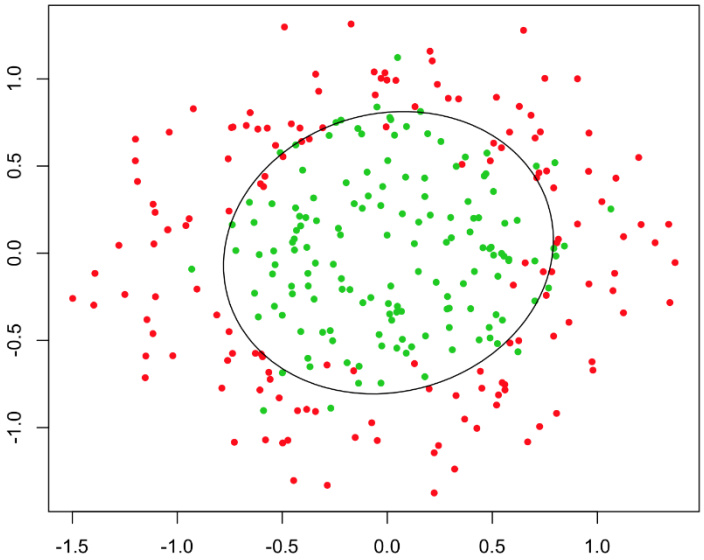

Figure 11 – Logistic regression

In the specific case of logistic regression (Figure 11), the constitutive function that exploits the sigmoid function can be written as:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \quad \rightarrow \quad 0 \leq h_\theta(x) \leq 1$$

The cost function deriving from this function exploits the logarithms to be able to dampen the effects of the exponential basis of the sigmoid function and therefore turns out to be different from the cost function of the linear regression.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \cdot log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \cdot log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right]$$

The equation for calculating the gradient descent is similar to that of linear regression, but obviously it will have a different constitutive equation inside.

$$\theta_J = \theta_J - \alpha \cdot \sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_J^{(i)}$$

1.11 Neural network

A model based on a neural network [10] as the name implies takes inspiration from the complexity of our brain. These types of models are among those that manage to obtain the best results and it is believed that by developing variants increasingly similar to the natural conformation of the human brain, the results of accuracy during training can also be improved. Neural networks are based precisely on the concept of networks (Figure 12) where each node is connected to many other nodes through links.

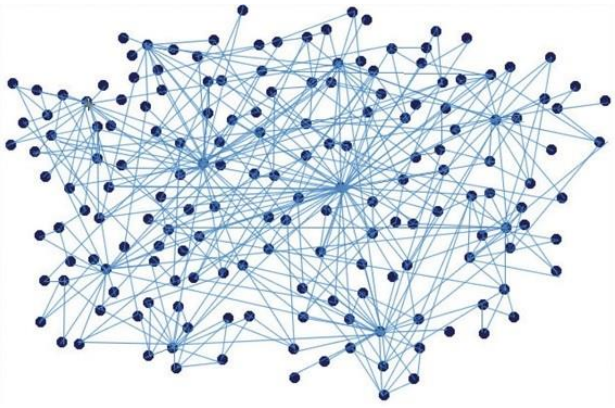

Figure 12 – Network with nodes and links

In this specific case the nodes are divided into layers (Figure 13), and the more layers there are, the more complex the neural network becomes.

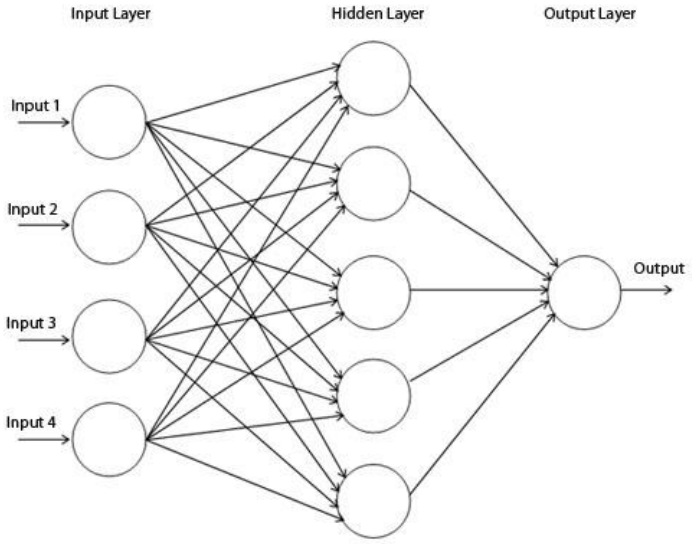

Figure 13 – Neural network with one hidden layer

A neural network is defined as deep (DNN) [11] [12] [13] [14] [15] when in addition to the input and output layers there are more than 1 hidden or transition layers (Figure 14).

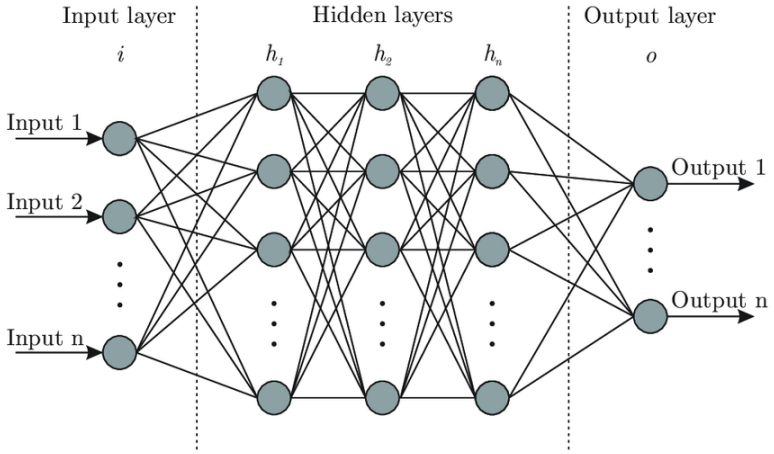

Figure 14 – Deep neural network with more than one hidden layers

The neural network remains in the category of supervised classification learning and allows to classify various categories, based on the number of nodes of the output layer, so if for example we are working on the CIFAR10 dataset which classifies 10 different objects then our deep neural network will have 10 nodes in the last layer, the output layer, and each node will correspond to a class. The input nodes, on the other hand, correspond to all the input parameters, while the nodes of the central layers do not have a real physical meaning and can be in variable numbers. Since the neural network is a model for classification, also in this case the sigmoid function is exploited and the resulting constitutive function is:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{k}\left[y_k^{(i)} \cdot log\left(h_\theta\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right) \cdot log\left(1 - h_\theta\left(x^{(i)}\right)\right)_k\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_l+1}\left(\theta_j^{(l)}\right)^2$$

In order not to make the treatment too heavy, other technical details on neural networks will not be added, but for a more detailed study, the publications mentioned at the end of the thesis can be consulted [10] [11] [12] [13] [14] [15].
Through various researches, it has been noticed that by increasing the number of hidden layers too much, we reach a point where the accuracy of the model does not grow, but on the contrary begins to decrease, therefore various variations to the classic DNN have been invented, including the CNN (convolutional neural network).

1.11.1 Convolutional neural network (CNN)

This type of neural network was created by taking inspiration from the animal visual cortex (Figure 15). The basic idea of a CNN lies in dividing the images into areas and extracting from each of them the most important features for evaluation purposes. CNNs are mainly used for the recognition of images and natural language. There are different CNNs model [16], among which the most important are:

- AlexNet
- VGG
- LeNet
- GoogLeNet
- ResNet

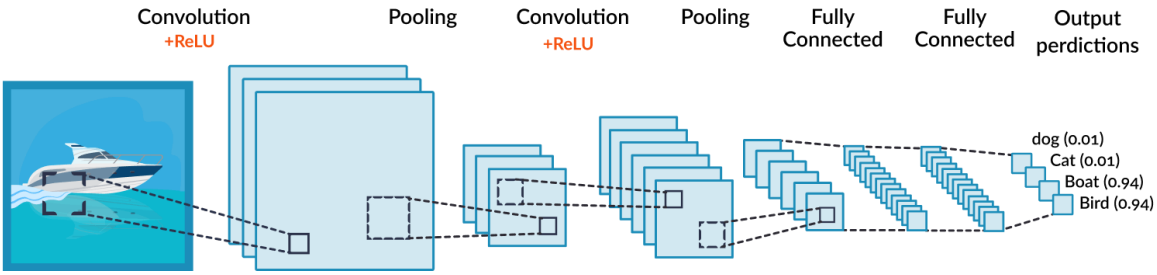

Figure 15 – Scheme of convolutional neural network

In some CNN variants, like ResNet (residual neural network) [17] [18] [19], not all the nodes of each layer are connected to the nodes of the next layer and not all the layers are taken into consideration at each cycle, this is because it has been noticed that even the neurons of the human brain do not have a perfection in the connections, but being a product of nature, they have many gaps and variations (Figure 16).

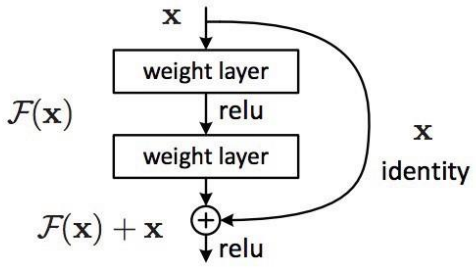

Figure 16 – Close up on ResNet model *[18]*

1.12 Problems and resolutions

There can be various types of errors during a machine learning process, we will see some of these and their resolutions in this paragraph.

1.12.1 Features scaling

The features are all the input parameters on which the training is based, but these variables do not always have values with similar orders of magnitude and this can often lead to computational problems when operations between use very large values and very small. To overcome this problem, feature scaling is used, that is, all parameters are rescaled on the same interval, as for example [0,1], so that each parameter, despite having different meanings, can be readable by the machine

with the same computational effort. In order to obtain this rescaling, the formula used is:

$$x_{new} = \frac{x}{N_x^\circ} \quad with \quad N_x^\circ = maximum\ value\ of\ x$$
$$e.g.: if\ 0 \leq x \leq 2000 \rightarrow N_x^\circ = 2000$$

1.12.2 Mean normalization

Mean normalization is an operation that is often carried out in parallel with the feature scaling in order to obtain not only values within a certain range, but also with a certain average value, therefore if for example the interval is [-1, 1] then the average value will be around 0.

$$x_{new} = x - x_{mean} \quad with \quad x_{mean} = mean\ value\ of\ x$$
$$e.g.: if\ 0 \leq x \leq 2000 \rightarrow x_{mean} = 1000$$

These feature scaling and mean normalization operations must be carried out for each individual feature taken into consideration before the start of the training process.

$$x_{new} = \frac{x - x_{mean}}{N_x^\circ}$$

1.12.3 Learning rate problems

The learning rate, as already specified in the paragraph dedicated to it, is the most important hyperparameter and consequently it is the one on which more attention must be paid.

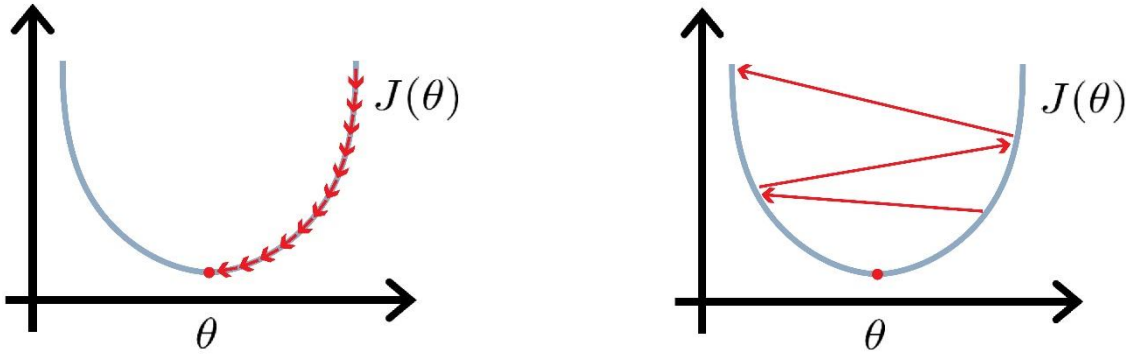

Figure 17 – Left: small learning rate; Right: large learning rate

With values that are too small (Figure 17 - left), there is a risk of not continuing forward during the training, remaining in a stall area or otherwise proceeding so slowly as to make the training useless. On the contrary, with too large values (Figure 17 - right) there is the risk of not being able to enter the minimum of the function in order to increase the accuracy and in the case of extreme values it is even possible to go to divergence and therefore bring the accuracy to values close to 0.

1.12.3.1 Learning rate finder

In order to find the correct value of the learning rate, there are various dedicated libraries for different programming languages, but the simplest method is to do a fairly long training, about 50 to 100 thousand epochs, during which the learning rate changes exponentially from very small values to very large values, the best choice is to try to vary the learning rate by at least 10 orders of magnitude throughout the training. Theoretically, if all the various parameters have been normalized then often the learning rate will be a value between 0.001 and 10, for this reason it is better to fully cover this range during the training. Once the test training is finished then it will be enough to take a look at the accuracy and error graph to find the maximum recommended learning rate.
As can be seen in Figure 18 for very small values of the learning rate, accuracy and error practically do not vary, however from a certain point onwards the error decreases up to a minimum and then goes up and diverges. This shows us that the actually useful learning rates are those included in the area of descent of the error, that is, from the initial plateau to the minimum point, beyond which the divergence begins. It is advisable to use a maximum learning rate value of about one order of magnitude lower than that found for the minimum point of the error, so as to be far enough from the divergence zone. So the learning should vary in the range between the start of the slope in the error up to an order of magnitude before the minimum point.

1.12.4 Training problems

During the training there may be problems related to the dataset in use and how the data are analyzed, consequently the training could lead to a model too adapted to the data on which it was trained or on the contrary too poorly adapted. These problems fall mainly into two categories:

- underfitting
- overfitting

Figure 18 – Learning rate finder

1.12.4.1 Underfitting

Underfitting (Figure 19) denotes a problem of high bias, that is, the error is too high and the model has not trained enough or is not suitable for a certain type of dataset, generally in the underfitting phase whether you test the model on training set that on the validation set in both cases the error will be high. To overcome this problem, it is often enough to increase the number of features by trying to make the model more precise and also increasing the training time so that the model has the time necessary to acquire the information.

1.12.4.2 Overfitting

When overfitting (Figure 19) occurs it means that there is a high variance problem, that is, the model has adapted specifically to the dataset on which it has trained, but would not be able to obtain the same results with a dataset never seen before. This error is typical and can be seen when for long periods of training the error for the training set continues to decrease, while that for the test set reaches an asymptotic level below which it does not drop or in any case decreases less quickly than the training set. If up to a few epochs before, the accuracy which in percentage divided the training set from the validation set was almost constant now instead tends to increase denoting a model too accustomed to the training set. To avoid this problem it is often enough to decrease the number of epochs, if excessive, or in most cases it is necessary to decrease the number of constraints and features in order to leave more margin of error to the model. For DNN models we can use dropout layers, which, for each epoch, will randomly remove certain features by setting them to zero and consequently the overfitting problem is solved.
In general, for underfitting and overfitting problems, it is possible to act on hyperparameters or features with regularization factors in order to create a reliable and robust model.

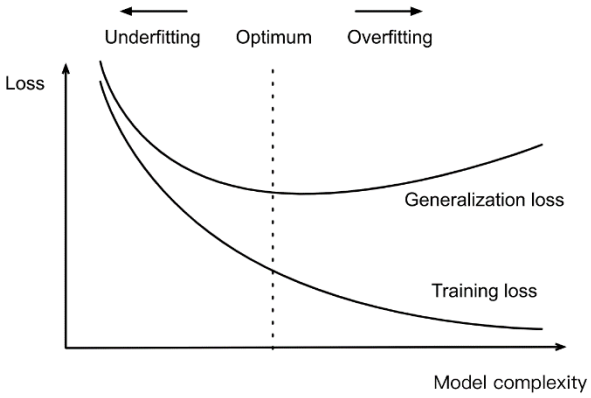

Figure 19 – Underfitting and overfitting

1.12.5 Random initialization

Often in machine learning it is necessary to have initial values for the variables with which to start and they are generally set to zero, but this choice is not always correct, since, especially in complex searches like these, it is better not to have initial symmetries in the model that they could lead the model to suddenly diverge or not to move at all from the initial null values. As an example, it can be thought that placing a ball on the top of a pointed pyramid is perfect and symmetrical, but given the unstable equilibrium, a small disturbance is enough to bring the model to complete ruin. Conversely, if the ball is placed on the bottom of a narrow and high basin, the ball can hardly go up the walls even in the presence of strong perturbations. So to avoid cases of singularities often the initial variables are placed random, this entails different results for each launch of the same training, but from a statistical point of view the variations are minimal and considering the average there will be acceptable and verified results.

2 – Python & TensorFlow

2.1 Programming languages

To be able to enter the world of machine learning it is necessary to use the programming languages. There are several, but those that are most useful for this purpose are those that allow efficient matrix calculation, since machine learning is mainly based on the management of large matrices. High-level languages are therefore more useful and most of the work is left to the interpreter. During this thesis two languages have been used mainly, that is:

- MATLAB
- Python

Each programming language has its own optimized libraries for mathematical calculation, but dedicated machine learning libraries have been developed which have often been released for various languages and therefore we will treat them as independent libraries:

- TensorFlow
- PyTorch
- Keras

2.2 MATLAB

MATLAB [20] is a programming language optimized for matrix calculation and for the representation of plots. Given the size of the dataset to be managed, writing in MATLAB improves code's formatting and understanding. Below there is an example of how normal equation (Chapter 1.6) can be calculated with MATLAB:

```
y = [1 2;3 4];
x = [5 6;7 8];
theta = inv(x' * x) * x' * y
```

Output:

```
theta =

    5.0000    4.0000
   -4.0000   -3.0000
```

To perform the same operation in a language like C, it would have taken many more lines of code and this explains why MATLAB is so useful in this field. However, as it is not an open-source language, it does not have many libraries optimized for machine learning and therefore for more in-depth studies, it is necessary to turn to the Python language.

2.3 Python

Python [21] [22] is a high-level programming language developed by Guido van Rossum in the 90s, it is object-oriented and open-source. It can be used for various functions: writing an application, sending an email, sending notifications through a Telegram bot, etc. It has been continuously improved and expanded, but since version 2.7 there has been a division that led to the creation of version 3.x. Currently only version 3.x is constantly updated and in many parts no longer compatible with version 2.7.
Being open-source it is one of the most used languages in the world, also in the scientific field [2] [1] for which there are various dedicated libraries, such as:

- Numpy - for matrix calculation and generic mathematical functions, it creates arrays to identify matrices
- Scipy - for mathematical analysis
- Matplotlib - for plotting graphs
- Pandas - for data analysis
- Scikit - for machine learning

Since Python is not optimized for matrix calculation as MATLAB, is not as immediate as the latter, but thanks to the aforementioned libraries it can manage the matrices. Here is the normal equation (Chapter 1.6) calculated with Python:

```
import numpy as np

y = np.array([[1,2],[3,4]])
x = np.array([[5,6],[7,8]])
```

```
x_tra = np.transpose(x)
x_inv = np.linalg.inv(x_tra @ x)

theta = x_inv @ x_tra @ y
```

Output:

```
theta =
array([[ 5.,  4.],
       [-4., -3.]])
```

2.3.1 Self parameter

The "self" parameter [23] in Python represents the instance of the class and allows to recall the attributes and methods of the class in question. Here is an example to better explain its function:

```
class example(object):

    def __init__(self,sun):
        self.sun=sun
        print(sun+2)
        print(self.sun+10)

    def cloud(self):
        self.rain=5*5+100
        self.rainbow=self.rain+1
        print(self.rain+self.sun)

x=5
example(x)
print('-------------')

y=example(-6)
print('=============')
y.cloud()
print('++++++++++++')
print(y.rain)
```

Output:

```
7
15
-------------
-4
4
=============
119
++++++++++++
125
```

Trough “self” parameter, as you can see, it is easier to perform calculations with variables.

2.4 TensorFlow

TensorFlow [24] [25] is a machine learning library released by Google in 2015. Since its release it has become the most used library in association with Python language, since they are both open source and TensorFlow allows you to automate training very well. TensorFlow is not always intuitive [26], but with many fundamental functions for the simplification of the code.
The logic on which TensorFlow is based are tensors [27], which are multidimensional matrices, and as the name suggests is a calculation based on the flow of tensors. Despite being used with Python, TensorFlow does not follow its logic [28] and bases its calculation on a graph system. When building variables in TensorFlow they are not instantly stored in RAM, but they become part of a graph, with all the variables and operations to be performed, and only once the run command is executed then actually the graph is calculated.
Even TensorFlow, like PyTorch, can be used with CUDA [29] and therefore use the computing power of Nvidia GPUs. The current version of TensorFlow is 2.x, but many codes are written with version 1.9 or earlier which differ in part with the 2.x.

2.4.1 TensorFlow functions

Here are some examples of functions in TensorFlow:

- Piecewise constant
- Exponentially decay
- Polynomial decay

2.4.1.1 Piecewise constant

```
tf.train.piecewise_constant(x, boundaries, values, name=None)
```

The function (Figure 20) uses a global variable `x` and divides the `values` among the various `boundaries`. An example:

```
global_step = tf.Variable(0, trainable=False)
boundaries = [100000, 110000]
values = [1.0, 0.5, 0.1]

learning_rate = tf.train.piecewise_constant(global_step,
boundaries, values)
```

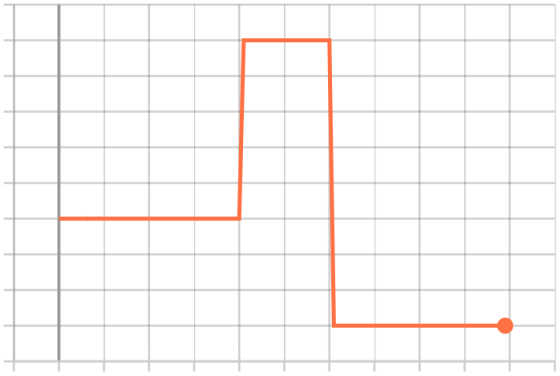

Figure 20 – An example of piecewise constant trend

2.4.1.2 Exponential decay

```
tf.train.exponential_decay(learning_rate, global_step,
decay_steps, decay_rate, staircase=False, name=None)
```

This function (Figure 21 - left) calculates the exponential decay [30] according to the formula:

$$output = learning\_rate \cdot decay\_rate^{\frac{global\_step}{decay\_steps}}$$

If `staircase` is `True` then `global_step / decay_step` becomes integer and there is a trend ad shown in (Figure 21 - right).

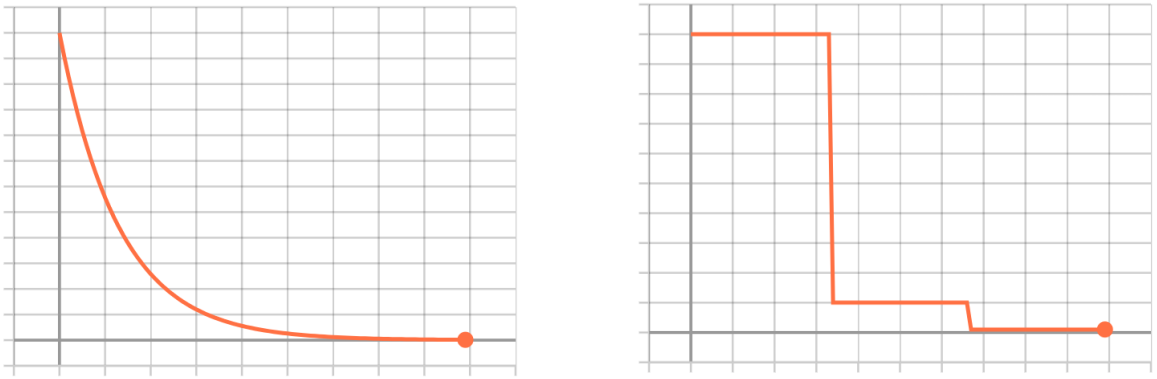

Figure 21 – Left: normal exponential decay; Right: staircase exponential decay

2.4.1.3 Polynomial decay

```
tf.train.polynomial_decay(learning_rate, global_step, decay_steps,
end_learning_rate=0.0001, power=1.0, cycle=False, name=None)
```

The polynomial decay (Figure 22 - left) is calculated as follows:

$$global\_step = min(global\_tep, decay\_step)$$

$$output = (learning\_rate\ - end\_learning\_rate) \cdot \left(1 - \frac{global\_step}{decay\_step}\right)^{power} + end\_learning\_rate$$

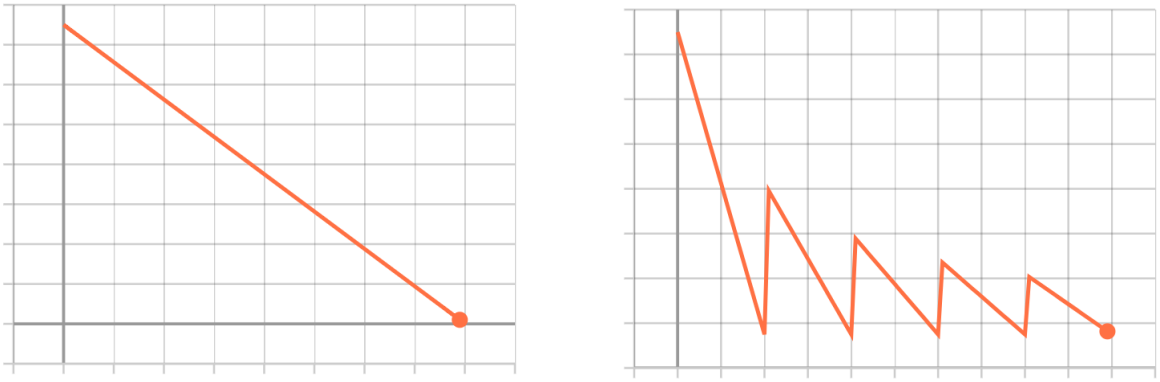

Figure 22 – Left: normal polynomial decay; Right: cycle polynomial decay

If `cycle` is `True` (Figure 22 - right) then `decay_steps` becomes:

$$decay\_steps = decay\_steps \cdot ceil\left(\frac{global\_step}{decay\_step}\right)$$

The `ceil` function returns the smallest integer value greater than the input.

2.4.2 TensorBoard

TensorBoard (Figure 23) is a tool that provides a graphic interface to TensorFlow and acts as a valid help tool for:

- Plotting accuracy and loss
- Displaying the TensorFlow graph
- Showing histograms and tensors values that change over time
- Displaying the data, such as images, texts, etc.

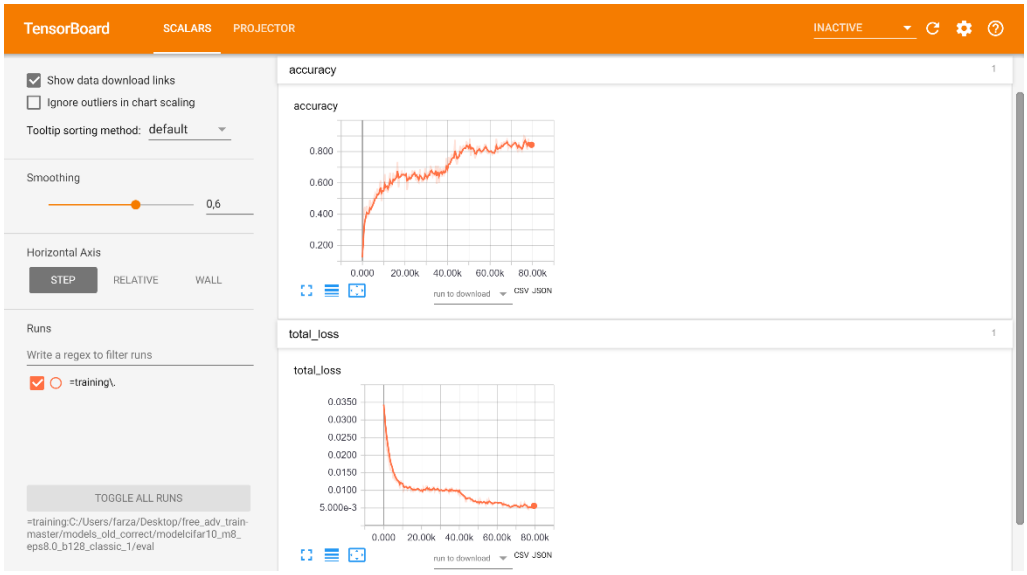

Figure 23 – An example of TensorBoard interface

2.5 PyTorch

It is a library [31] released by Facebook for machine learning. It is more intuitive than other libraries in the same field and also allows for ease of use, but having been released later than TensorFlow many codes are currently written only for TensorFlow, for which more detailed documentation is generally found. For

managing arrays, PyTorch uses a class called Tensor with which it creates multidimensional arrays whose operations can also be performed on CUDA-capable [29] Nvidia GPU.

2.6 Keras

Keras is a library [32] for machine learning currently written only in Python language. It is mainly used for a rapid realization of DNNs. Compared to other libraries it is at a higher level, allowing a higher level of abstraction. During the course of this thesis it was initially used to test basic machine learning codes for Python. It can be used in conjunction with TensorFlow.

Part II

-

Research Field

3 – Adversarial training

3.1 Why do we need adversarial training?

Adversarial training [33] [34] [3] [13] is a branch of machine learning that deals with creating robust models against adversarial attacks. For years the training has only been concerned with looking for models that could achieve ever greater accuracy, up to reaching such high standards as to exceed even the human being in certain datasets, but as in any field of computer science there can be a malicious attack from the outside with the intention of carrying out illegal actions. For example, machine learning [35] is used in banking to easily recognize handwritten numbers, but if a model is attacked it would lead to incorrect evaluation of the numbers with serious consequences. In other cases, facial, voice or fingerprint recognition is used to unlock certain services and even in this case an external attack can cause damage. To counter these attacks we need robust models with a good accuracy. The problem was that the real robustness of these models was not known, that is, whether these models were truly able to respond adequately in case of random or targeted variations. At the beginning of the 2010s, after various researches, this new branch was born which is called adversarial training.

3.2 Adversarial examples

The basic logic is to create imperceptibly modified examples [36] and see if the model can recognize them or not. If the examples were modified with a random logic then there would be no problems, since the model would fail, but a human being would also fail and therefore the problem does not exist, however in the event of a malicious attack some examples could be modified in order to mislead a classic model, but without any obvious variation for the human eye. As can be seen in (Figure 24) the two images (original and modified by attack) are identical to the human eye, but every single pixel has been modified according to the plot that can be seen in the middle.

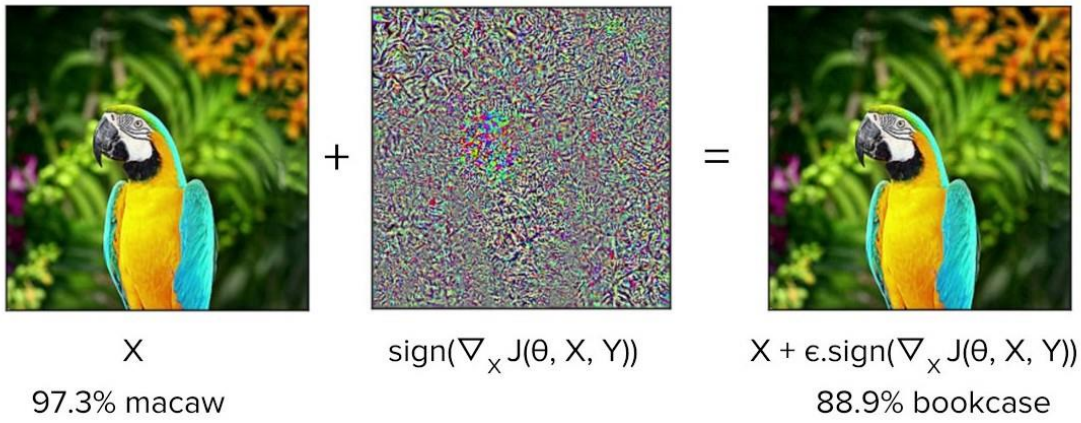

Figure 24 – An example of adversarial attack: a macaw is misclassified as a bookcase *[36]*

This means that a model can be attacked very easily without obvious external signs, in reality the situation is more complex than it seems because just as two identical images to the human eye deceive a model, on the contrary there are completely modified images (Figure 25) that the model succeeds to recognize.

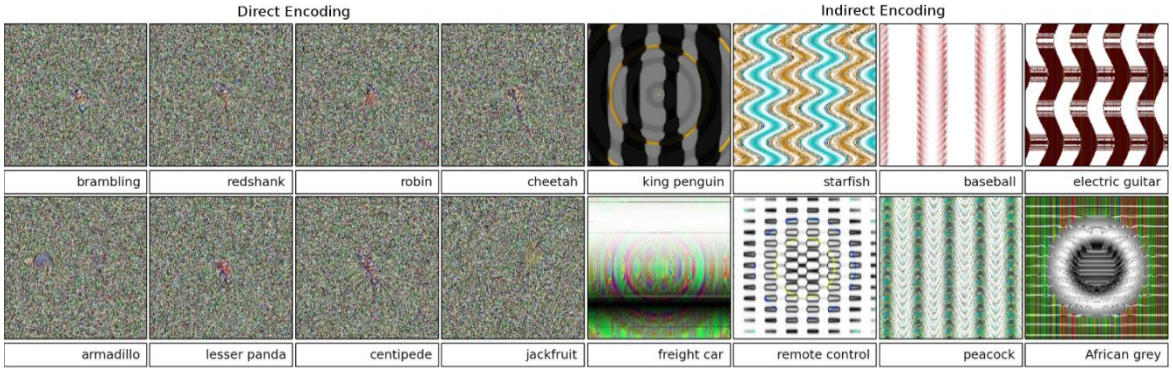

Figure 25 – Some images correctly identified by the machine

3.3 Adversarial attacks

A model can learn to recognize images, but in a completely different way from how a human being does it, so there are various attack logics. Some attacks are based on varying a single pixel (Figure 26) [37], others on some features of the image, but the most common case is to calculate the division line that distinguishes one class from another.

Figure 26 – Examples of single pixel attack *[37]*

In a simple example with two classes the decision boundary looks like in the Figure 27. A targeted attack would take all the borderline cases, so in the case of an image it would take the pixels closest to the edge of the line, and vary them just enough to make them cross the line (Figure 28 - left) in order to completely distort the classification without actually changing much. All this in a complex image creates two identical images, but really different for a model.

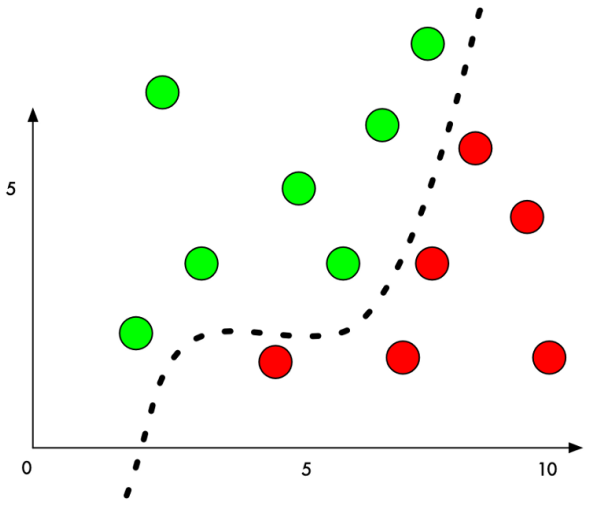

Figure 27 – Decision boundary between two classes

In some cases the pixels do not change, but it is the separation line that distinguishes the classes that moves (Figure 28 - right) changing the boundary between the classes themselves, equally leading to a misclassification.

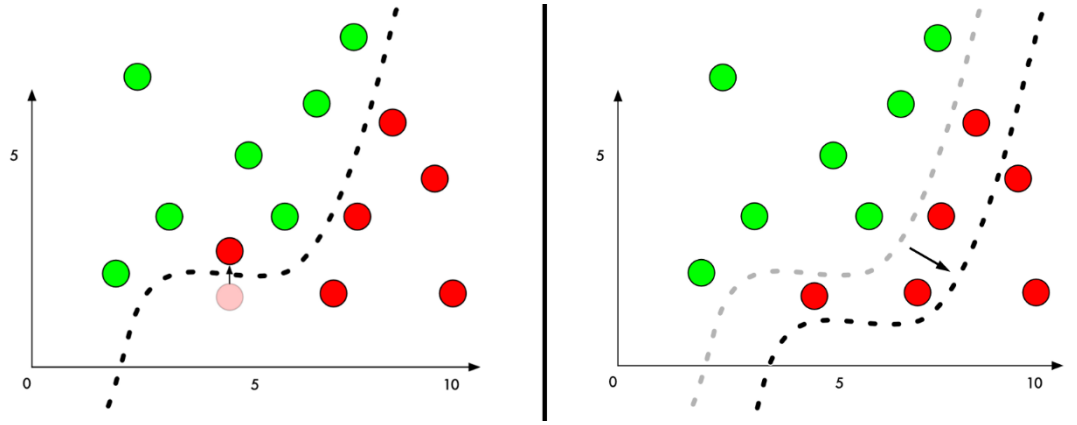

Figure 28 – Two ways of fooling a classifier – Left: pixels moving; Right: line moving

Attacks are mainly divided into 2 categories:

- White-box
- Black-box

3.3.1 White-box attacks

This is the easiest type of attack [38] to perform. It is based on the knowledge of the internal structure of the model to be attacked (Figure 29), in this way the attack is more specific and the damage caused increases. The white-box attack was used during this thesis.

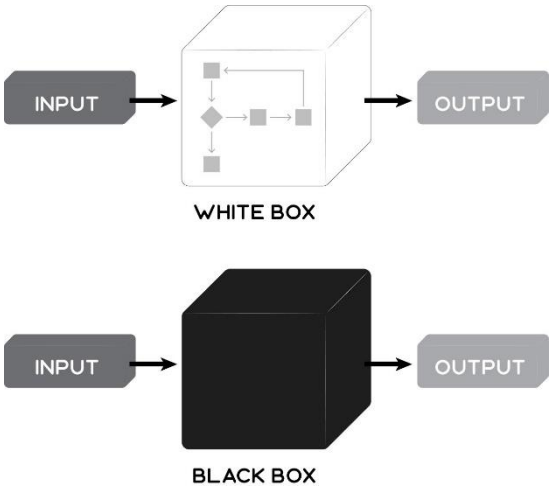

Figure 29 – White-box VS black-box attack *[38]*

3.3.2 Black-box attacks

A black-box attack [39] means that the attacker does not know the internal structure of the model and tries to create a blind attack (Figure 29). Obviously these types of attacks are less effective but applicable to multiple models without significant variations. In 2017, a challenge was realized [38] based on this type of attack to verify the resistance of the model created.

3.4 Training against adversarial attacks

Various techniques can be used to create models resistant to enemy attacks:

- Data augmentation
- Defensive distillation
- Second model control

Once the logic between attack and defence [40] [41] of the models is understood, the main problem is to train robust models in reasonable time and not too long compared to the classic training algorithms.

3.4.1 Data augmentation

This is the main technique to counter attacks by training models on already attacked images [42] and not only on correct images, significantly increasing the accuracy against attacks, sometimes from 0% it can also reach an accuracy of about 50% [43] [44], but of course this means that the accuracy on normal images decreases and often falls below the accuracy level of a human being [45].

3.4.2 Defensive distillation

This type of defence consists of two phases in which training takes place through a process called distillation and prevents the model from adapting too much to the data it is examining. This technique has been studied in particular in some papers [46] and works very well on some types of specific attacks, but it is not robust for all types of attacks.

3.4.3 Second model control

The second model control [47] uses another model that is trained on the main neural network and its internal characteristics so that it can predict whether the

example analyzed is adversarial or not. In practice, this technique uses an "external guard" logic that controls the whole process to verify its effective operation. It is a robust technique, but still under study to actually verify its effectiveness [48] and complexity.

3.5 Adversarial libraries

To allow the attacks there are various libraries, among which the main ones are:

- Cleverhans
- Foolbox

Both libraries [49] [50] have been tested to create attacks, using both PyTorch and TensorFlow. The logic is similar for both, you have to convert the model to the specific model of the attack library, choose the parameters and finally decide the type of attack. Once the structure is created, the model is attacked and its accuracy is tested.
Below there are the basic steps and results (Figure 30) for the Foolbox [51] library, the one mainly used during this thesis. It is a simple example with Python and PyTorch [52].

```
import foolbox
import numpy as np
import torchvision.models as models

# get model in PyTorch
model = models.resnet18(pretrained=True).eval()
preprocessing = dict(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225], axis=-3)

# create model in Foolbox
fmodel = foolbox.models.PyTorchModel(model, bounds=(0, 1),
num_classes=1000, preprocessing=preprocessing)

# get a batch of images and labels and print the accuracy
images, labels = foolbox.utils.samples(dataset='imagenet',
batchsize=16, data_format='channels_first', bounds=(0, 1))
print(np.mean(fmodel.forward(images).argmax(axis=-1) == labels))

# apply the attack
attack = foolbox.attacks.FGSM(fmodel)
adversarials = attack(images, labels)
```

```
print(np.mean(fmodel.forward(adversarials).argmax(axis=-1) ==
labels))

# print the first image with matplotlib
[…]
```

Output:

```
0.9375
0.0
```

Figure 30 – Adversarial example generated using Foolbox *[49]*

3.6 Free Adversarial Training (FAT)

During the thesis, various papers [53] [54] were evaluated to verify the state of the art in this field and, in the end, Free Adversarial Training (FAT) [55] was chosen as the baseline code for the experiments.
The FAT code is based on a deep neural network (DNN) and the structure is based on a residual neural network (ResNet) model. The attacks are generated with a projected gradient descent (PGD) [56] logic and the various hyperparameters can be decided. There are 2 datasets that can be used: CIFAR10 [7] and CIFAR100 [7]. In the first part of the thesis the results of the original paper were replicated and afterwards the changes for the improvements were applied.
The code is made up of various files:

- free_train.py – the main code that allows you to train your adversarial examples
- free_model.py – the modified ResNet model used by FAT with its convolutional layers

- config.py – a configuration code that allows you to simply change the training and attack inputs from the command line
- cifar10_input.py – the preparation code of the CIFAR10 dataset
- cifar100_input.py – the preparation code of the CIFAR100 dataset
- multi_restart_pgd_attack.py – the code that creates adversarial examples for both training and evaluation tests

The main FAT algorithm on which its code is based can be seen in (Figure 31)

Require: Training samples X, perturbation bound ϵ, learning rate τ, hop steps m
Initialize θ
$\delta \leftarrow 0$
for epoch $= 1 \dots N_{ep}/m$ **do**
  for minibatch $B \subset X$ **do**
    for i $= 1 \dots m$ **do**
      Update θ with stochastic gradient descent
        $g_\theta \leftarrow \mathbb{E}_{(x,y)\in B}[\nabla_\theta\, l(x+\delta, y, \theta)]$
        $g_{adv} \leftarrow \nabla_x\, l(x+\delta, y, \theta)]$
        $\theta \leftarrow \theta - \tau g_\theta$
      Use gradients calculated for the minimization step to update δ
        $\delta \leftarrow \delta + \epsilon \cdot \text{sign}(g_{adv})$
        $\delta \leftarrow \text{clip}(\delta, -\epsilon, \epsilon)$
    end for
  end for
end for

Figure 31 – FAT algorithm *[55]*

To better understand the algorithm, here is the list of variables used:

- X – training samples
- ε – perturbation bound
- τ – learning rate
- θ – learning coefficient
- δ – actual perturbation
- N_{ep} – number of epochs
- B – minibatch
- g – gradient descent
- x – single image
- y – single label
- m – hop step

The “m” parameter is the heart of this algorithm also called Free-m. The m allows to repeat the perturbation several times for each single minibatch in order to obtain adversarial examples within the limits set by the parameter “ε”. By keeping “m = 1” a normal training is obtained, while increasing “ε” out of proportion would cancel the purpose of the research itself because the images would be too altered even for the human eye.

4 – Fast training

4.1 Why fast?

We need a fast training to meet an ever increasing demand for large databases to be managed in real time. For example, for the development of self-driving machines, a machine learning system is needed for instantly recognizing road signs, traffic lights or obstacles and must be able to constantly adapt to changes. In the case of websites that manage large databases, such as: Google, YouTube, Facebook, etc., machine learning allows you to manage the constant flows of incoming data, trying to provide the user with the fastest possible response. Just as the world around us goes faster and faster, such as means of transport, machine learning must also adapt to ever faster response times.
Fast training includes all the techniques used in machine learning to speed up training times. With current processors, CPUs and GPUs, computation times for full training can last from a few hours to several days.
To increase the speed there are various ways:

- in some cases specific techniques are used for the type of model considered, for example only DNNs [57] [58] [59] or even more specifically only ResNet are examined
- in other cases generic modifications are implemented that are applicable to almost all models

4.1.1 Fast training techniques

The specific techniques are different from code to code and therefore each research work shows its application. For example, for DNNs there are often changes to certain layers [55] [54] , but these techniques have not been covered in this thesis.
On the other hand, generic techniques mainly include the changes to be made to hyperparameters and more specifically to the learning rate [60], since variable

values can give better results in terms of performance compared to constant values. Among the various state-of-the-art fast training methodologies proposed in the literature, the most important in this regard were the following:

- 1 Cycle policy
- Cyclical policy
- Warm restarts

4.2 1 Cycle policy

The 1 cycle policy [61] is based on the principle of varying the learning rate and other hyperparameters during the training course to obtain fast training. As the name implies, the basic idea is to apply a single cycle to this hyperparameters throughout the training. High learning rate values with the 1 cycle policy lead to the reduction of other regularization values since the 1 cycle policy is a regularization in itself.

4.2.1 1 Cycle – Learning rate

After finding the maximum learning rate through the learning rate finder, (Figure 32) an initial value equal to 1/10 of the maximum is set, after which for about 90% of the total training (90% of the total epochs) there will be a complete cycle from 1/10 of the maximum, up to maximum, to then return to 1/10 of the maximum. In the last epochs, equal to about 10% of the total, there will be a rapid drop in the learning rate up to a value equal to 1/1000 of the maximum. The logic behind this cycle is:

- start with a fairly high and acceptable learning rate
- continue to increase it to descend more rapidly into the local minimum or to find deeper minimums
- decrease the learning rate again to enter more deeply into the minimum found
- drastically reduce the learning rate to try to reach the local minimum point

If the final part of drastic reduction lasted too many epochs, there would be overfitting, while if it lasted too little, the accuracy would remain too low.
An example of code to create the learning rate shape using the TensorFlow polynomial decay function:

```
lr_deeper=train_steps*0.9
lr_max=0.15
```

```
lr_1=tf.train.polynomial_decay(-lr_max*1.8, global_step,
lr_deeper/2, 0.0, 1.0)
lr_2=tf.train.polynomial_decay(lr_max*1.8, global_step, lr_deeper,
0.0, 1.0)
lr_3=tf.train.polynomial_decay(lr_max/1000, train_steps-
global_step-1, train_steps-lr_deeper-1, lr_max/10, 1.0)

learning_rate=lr_1+lr_2+lr_3
```

Here is the meaning of the variables:

- lr_max – maximum learning rate found with learning rate finder
- lr_deeper – the moment when the cycle starts the last piece between `lr_max/10` and `lr_max/1000`
- train_steps – total training time
- global_step – training steps counter
- lr_1, lr_2, lr_3 – the 3 parts of the cycle that added together give the complete 1 cycle policy shape for the learning rate

Figure 32 – 1 Cycle policy *[61]* learning rate

4.2.2 1 Cycle – Momentum

The 1 cycle policy does not apply only to the learning rate, but also to the momentum with an opposite shape. In this way, the regularization carried out on the learning rate is not dampened by the momentum, but on the contrary it is strengthened.
There is a maximum recommended momentum value of 0.95, while the minimum should be 0.85. In the final part of the training, while the learning rate decreases

rapidly, the momentum remains fixed at the maximum value of 0.95 (Figure 33).

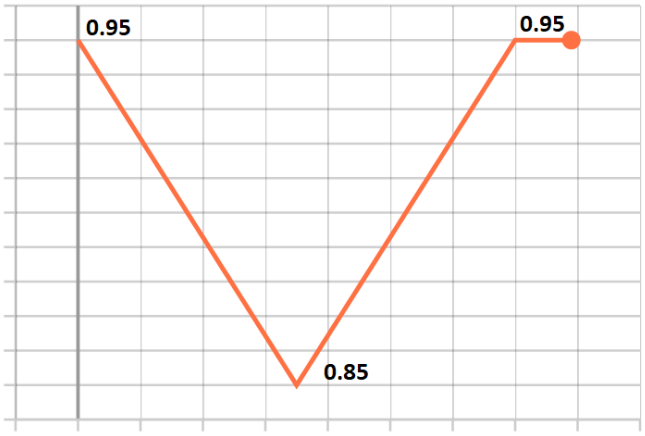

Figure 33 – 1 Cycle policy *[61]* momentum

4.2.3 1 Cycle – Other hyperparameters

The batch size should be set to the highest possible value to fit in the available memory.
The epochs depend on the accuracy that you want to achieve and therefore it is at the discretion of the programmer.
The weight decay must be tested with various values by running the learning rate finder each time, the correct weight decay to choose is the one that allows a higher maximum learning rate (Figure 34).

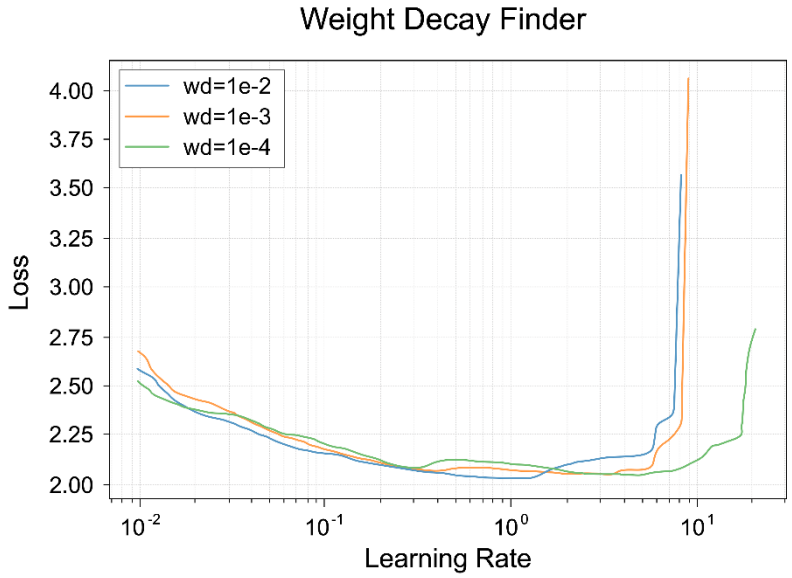

Figure 34 – In this case the weight decay of 10^{-4} is chosen because it corresponds to a higher maximum learning rate *[61]*

The remaining hyperparameters have to be chosen by running the learning rate finder each time, the important thing is that all the hyperparameters are set when you start the learning rate finder for the last time and choose the maximum learning rate, since training must take place in the same learning rate finder conditions.

4.3 Cyclical policy

Cyclical policy [62] is similar to 1 cycle policy, with the difference that the cycle is repeated several times, always oscillating between the same maximum and minimum values (Figure 35). This policy can be useful if the model has many local minimum points and therefore using a cyclical learning rate allows training to seek deeper minimums to achieve higher accuracy.

Figure 35 – A triangular cyclical policy *[62]*

4.3.1 Cycle length

The length of each single cycle is calculated as a multiple of an epoch. It is recommended to use length values equal to 4 - 20 times an epoch. It has been shown that for values within this range the optimal result is obtained. It is however advisable to do a training with at least 3 - 5 cycles to obtain an evident improvement. Increasing the number of cycles too much would eliminate the usefulness of the cycle itself, the training would not have time to adapt to the variation of the learning rate.

4.3.2 Cycle boundary values

The maximum and minimum values of the cycle must be chosen carefully because the success of the training depends on them. In both cases it is necessary to use the graph produced by the learning rate finder (Figure 36), which must be run before the final training. The maximum learning rate is found exactly as for the 1 cycle policy, i.e. the minimum point of the loss corresponds to the limit and the maximum learning rate must be chosen before this limit. For the minimum, instead, you choose a value in the descent zone of the loss, therefore from the moment in which the initial plateau ends forward.

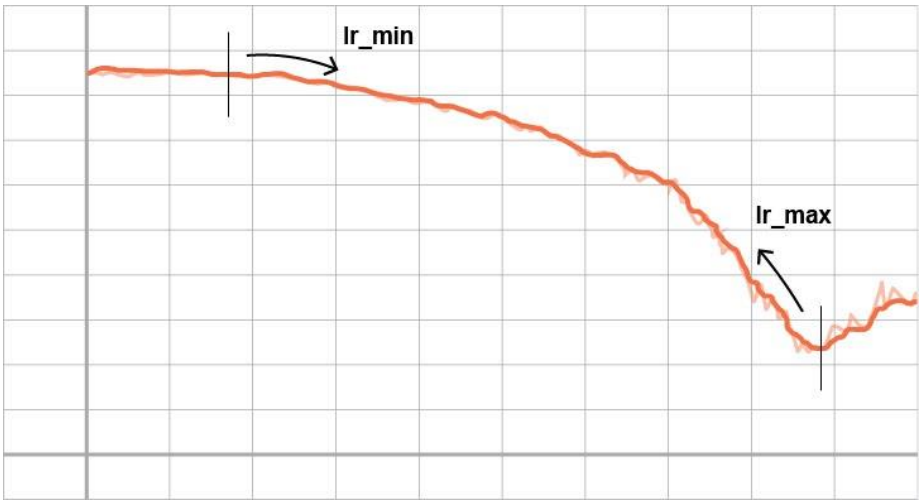

Figure 36 – Learning rate boundary on the loss plot for the cyclical policy *[62]*

In some cases the cycles are repeated with the same length but the maximum value decreases to allow you to search deeper in the local minimums, such as the decreasing triangular cycles of Figure 37.

Figure 37 – Cyclical policy with fixed lower boundary

4.4 Warm restarts

The warm restarts [63] [64] are also based on a cyclical policy, but as the term itself says, there are sudden restarts from the minimum to the maximum value (Figure 38). This phenomenon leads to instantaneously varying the learning rate and then start a long descent again, so as to find, as in the cyclical policy case, deeper minimums.
The warm restarts are always performed on the learning rate and can be performed with various shapes:

- linear
- sinusoidal
- trapezoidal

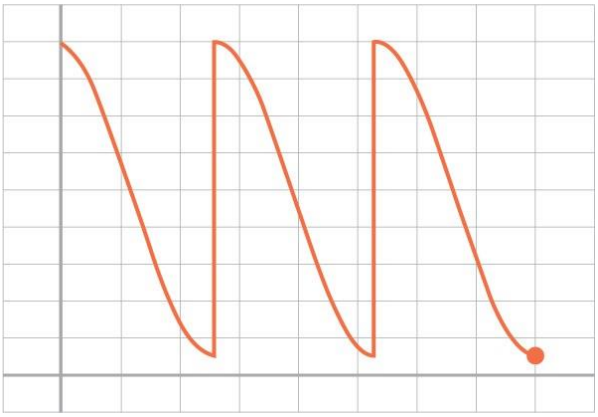

Figure 38 – Sinusoidal warm restarts *[63]*

The warm restarts can have multipliers that make the progress accordion-like during the training (Figure 39 - left) or the restarts can be at different values, gradually decreasing (Figure 39 - right).

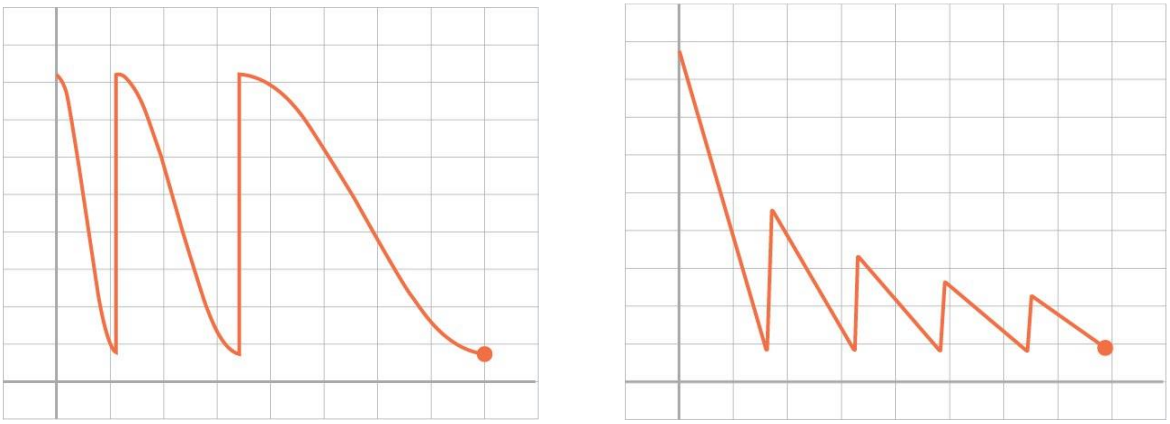

Figure 39 – Left: accordion-like warm restarts *[63]*; Right: decreasing warm restarts *[64]*

4.5 Other implementations

By changing the shapes of the hyperparameters or by mixing the techniques already mentioned, it is possible to obtain new trends in the learning rate, sometimes even more effective than the originals (Figures 40 – 41 – 42).

Figure 40 – 1 Cycle truncated

Figure 41 – 1 Cycle and half

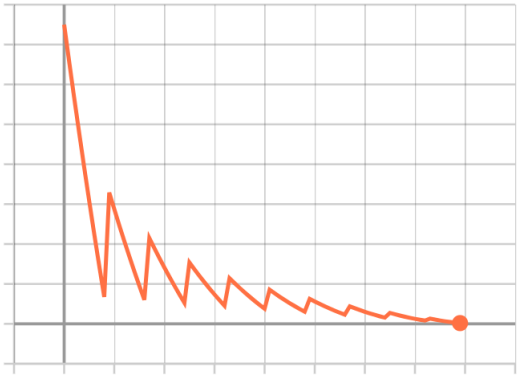

Figure 42 – Linear decay warm restarts

5 – Fast adversarial training

5.1 Super model

The aim of this thesis was to demonstrate that fast training techniques can also be applied to adversarial training, obtaining significant improvements in this case too. A new generation of DNN models is therefore created, which are both robust and fast: the super models. In future the super models will be normal, but we have to first demonstrate their feasibility. In order to achieve this, the Free Adversarial Training code was used, i.e. a DNN ResNet, with 2 different datasets: CIFAR10 and CIFAR100.

5.2 FAT results

In order to improve the FAT, it was necessary to first test the code on the calculator in use to verify its performance and after checking the results of the paper [55] the fast training techniques were tested. Following (Figures 43 - 44) are the results obtained with the training of the original FAT code.
As can be seen in the two graphs, the results are reported for both the CIFAR10 and the CIFAR100. It was possible to expect that the accuracy of the model trained on the CIFAR100 was lower than that of the CIFAR10, since there are more images to analyze there is a greater risk of error. In addition to the accuracy plot, the loss graph was also reported, in which the disparity between CIFAR10 and CIFAR100 is noted again. In both cases, the smoothed performance of the training was also reported to have a clearer view of the results.

Final accuracy results:

- CIFAR10 ➔ accuracy: 84.34%
- CIFAR100 ➔ accuracy: 59.89%

Final loss results:

- CIFAR10 ➔ loss: 0.00562
- CIFAR100 ➔ loss: 0.01459

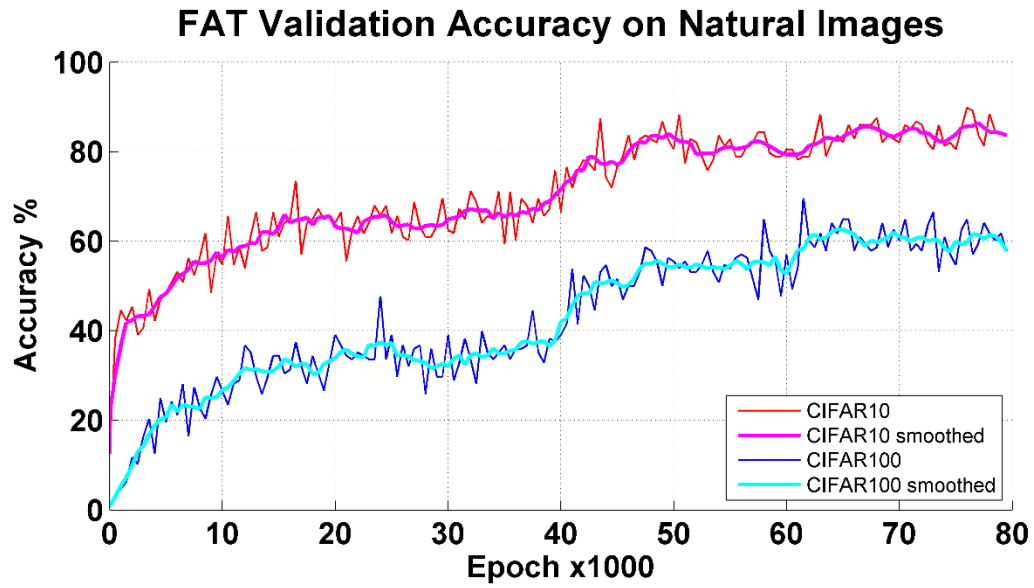

Figure 43 – Original FAT *[55]* accuracy on natural images

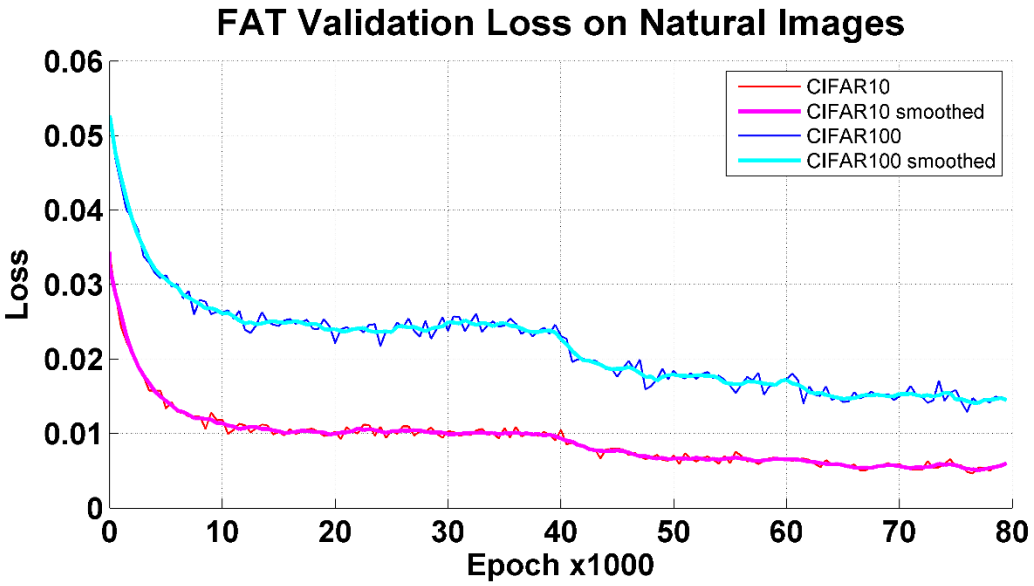

Figure 44 – Original FAT *[55]* loss on natural images

5.3 Hyperparameters tested

The original learning rate was a 3-steps function (Figure 45) with the following division:

- epochs = 0 – 40000 ➔ learning rate = 0.1
- epochs = 40000 – 60000 ➔ learning rate = 0.01
- epochs = 60000 – 80000 ➔ learning rate = 0.001

After using the learning rate finder was found a maximum value equal to:

- CIFAR10 ➔ maximum learning rate = 0.1 – 0.15
- CIFAR100 ➔ maximum learning rate = 0.1 – 0.12

Therefore a maximum learning rate higher than that the original FAT was used in the simulations.

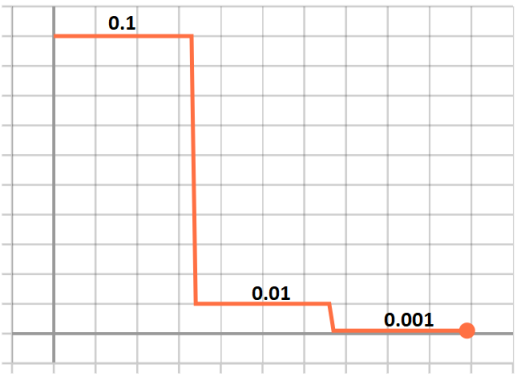

Figure 45 – FAT's 3-steps learning rate *[55]*

The momentum values used were two:

- 1 Cycle ➔ momentum = 0.85 – 0.95
- Constant ➔ momentum = 0.90

For the first part of simulations the 1 cycle momentum was used to test the 1 cycle policy code [61], but then the momentum was fixed on a constant value of the original FAT to get a clearer picture of how all the fast training techniques influence the learning rate and therefore the speed with the same momentum.

Based on the other regularisations, the value of the weight decay has been set at 0.0002.
The batch size value has been set to 128, due to computational limits of the calculator.
The remaining FAT parameters that are specific to adversarial training have not been changed because the aim was not to obtain a more robust model, but a faster model with the same robustness.

5.3.1 Learning rate's shapes summary

There is a summary (Figures 46 – 47 – 48 – 49 – 50 – 51 – 52 – 53) of all the shapes used during this thesis.

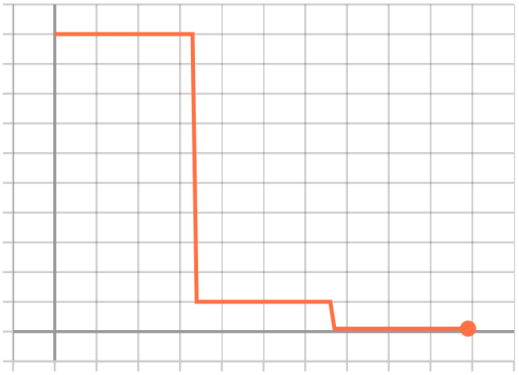

Figure 46 – 3 Steps

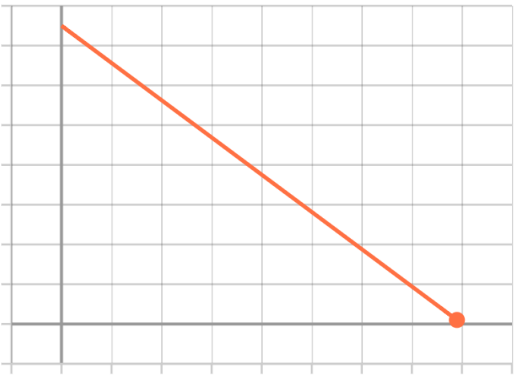

Figure 47 – Linear Decay

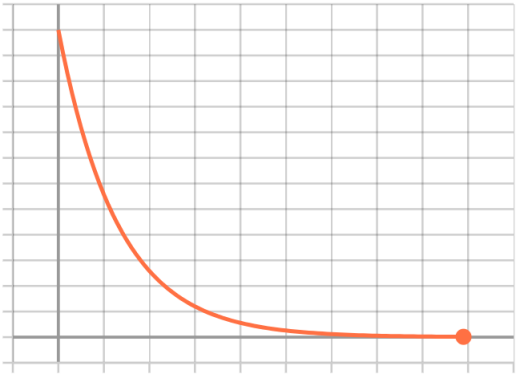

Figure 48 – Exponential Decay

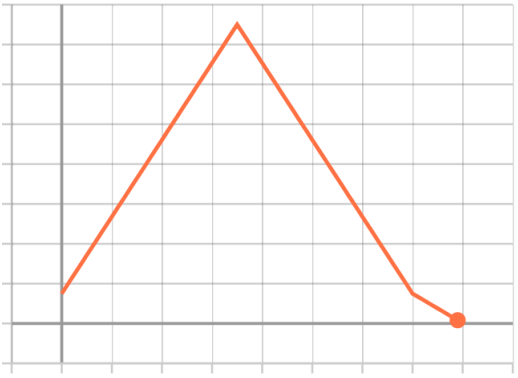

Figure 49 – 1 Cycle

Figure 50 – 1 Cycle & Half

Figure 51 – 1 Cycle Truncated

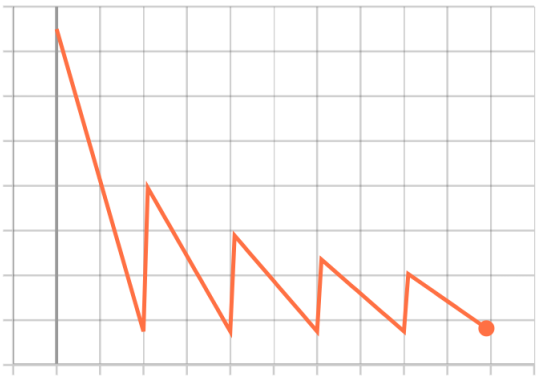

Figure 52 – Warm Restarts

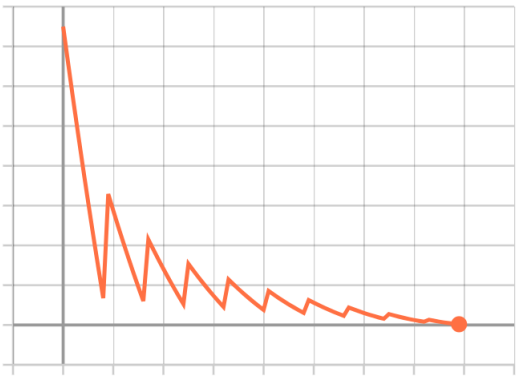

Figure 53 – Linear Decay Warm Restarts

5.3.2 Momentum's shapes summary

Here (Figures 54 – 55) are the two types of momentum used.

Figure 54 – 1 Cycle

Figure 55 – Constant

5.4 Simulations

All simulation codes and techniques are summarized in Tables 1 – 2. Here is the name's encoding:

D_F2_E_M

- D – Dataset
- F – Fast training technique
- 2 – If present it means that also the momentum is 1 Cycle
- E – Epochs
- M – Maximum learning rate

Name	Dataset	Epochs (x1000)	Momentum policy MAX - min	Learning rate MAX min	Learning rate policy
10_FAT_80_0.10	**CIFAR10**	**80**	**Constant 0.90**	**0.1 0.001**	**3 Steps**
10_1cyc_80_0.10	CIFAR10	80	Constant 0.90	0.1 0.0001	1 Cycle
10_1cyc2_60_0.10	CIFAR10	60	1 Cycle 0.95 – 0.85	0.1 0.0001	1 Cycle
10_1cyc2_40_0.15	CIFAR10	40	1 Cycle 0.95 – 0.85	0.15 0.00015	1 Cycle
10_1cyc2_40_0.12	CIFAR10	40	1 Cycle 0.95 – 0.85	0.12 0.00012	1 Cycle
10_1tru2_40_0.15	CIFAR10	40	1 Cycle 0.95 – 0.85	0.15 0.00015	1 Cycle Truncated
10_lin2_40_0.15	CIFAR10	40	1 Cycle 0.95 – 0.85	0.15 0.00015	Linear Decay
10_1half_50_0.15	CIFAR10	50	Constant 0.90	0.15 0.00015	1 Cycle & Half
10_warm_50_0.15	CIFAR10	50	Constant 0.90	0.15 0.015	Warm Restarts
10_walin_40_0.15	CIFAR10	40	Constant 0.90	0.15 0.00015	Linear Decay Warm Restarts
10_exp_45_0.12	CIFAR10	45	Constant 0.90	0.12 0.00012	Exponential Decay
10_const_40_0.15	CIFAR10	40	Constant 0.90	0.15 0.15	Constant
10_exp_40_0.15	CIFAR10	40	Constant 0.90	0.15 0.00015	Exponential Decay
10_warm_40_0.15	CIFAR10	40	Constant 0.90	0.15 0.0015	Warm Restarts
10_1cyc_40_0.15	CIFAR10	40	Constant 0.90	0.15 0.00015	1 Cycle

Table 1 – Summary of CIFAR10 simulations

Name	Dataset	Epochs (x1000)	Momentum policy MAX - min	Learning rate MAX min	Learning rate policy
100_FAT_80_0.10	**CIFAR100**	**80**	**Constant 0.90**	**0.1 0.001**	**3 Steps**
100_walin_60_0.12	CIFAR100	60	Constant 0.90	0.12 0.00012	Linear Decay Warm Restarts
100_exp_45_0.12	CIFAR100	45	Constant 0.90	0.12 0.00012	Exponential Decay
100_lin_40_0.12	CIFAR100	40	Constant 0.90	0.12 0.00012	Linear Decay
100_1cyc_40_0.12	CIFAR100	40	Constant 0.90	0.12 0.00012	1 Cycle
100_exp_40_0.12	CIFAR100	40	Constant 0.90	0.12 0.00012	Exponential Decay
100_warm_40_0.12	CIFAR100	40	Constant 0.90	0.12 0.0012	Warm Restarts
100_walin_40_0.12	CIFAR100	40	Constant 0.90	0.12 0.00012	Linear Decay Warm Restarts
100_const_40_0.12	CIFAR100	40	Constant 0.90	0.12 0.12	Constant

Table 2 – Summary of CIFAR100 simulations

5.4.1 Natural images results

The final results obtained with the simulations are summarized in Figures 56 – 57. The most important simulations have been represented in Figures 58 – 59 – 60 in order to better see the trends compared to the original FAT.
The simulations were performed with various epochs values to show the differences even if the algorithm used is the same. Keep in mind that with the calculator used 10000 epochs correspond to about 5 hours of simulation, so for example the original FAT lasts about 40 hours of calculation, that is almost 2 days. If you can even halve the number of epochs, the time gain is significant.

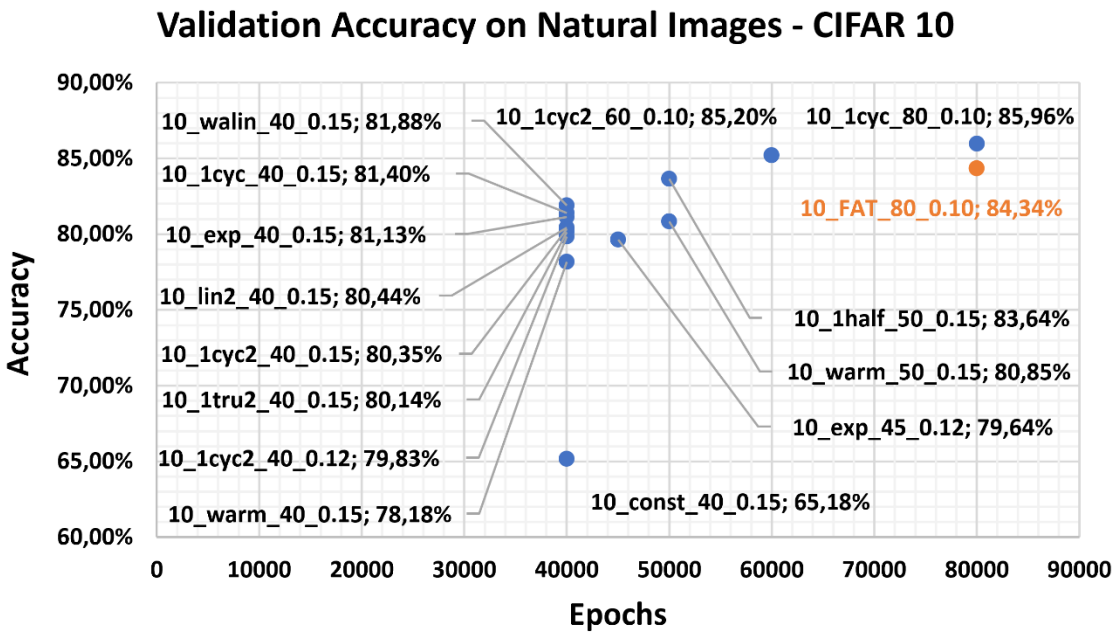

Figure 56 – CIFAR10 accuracy on natural images

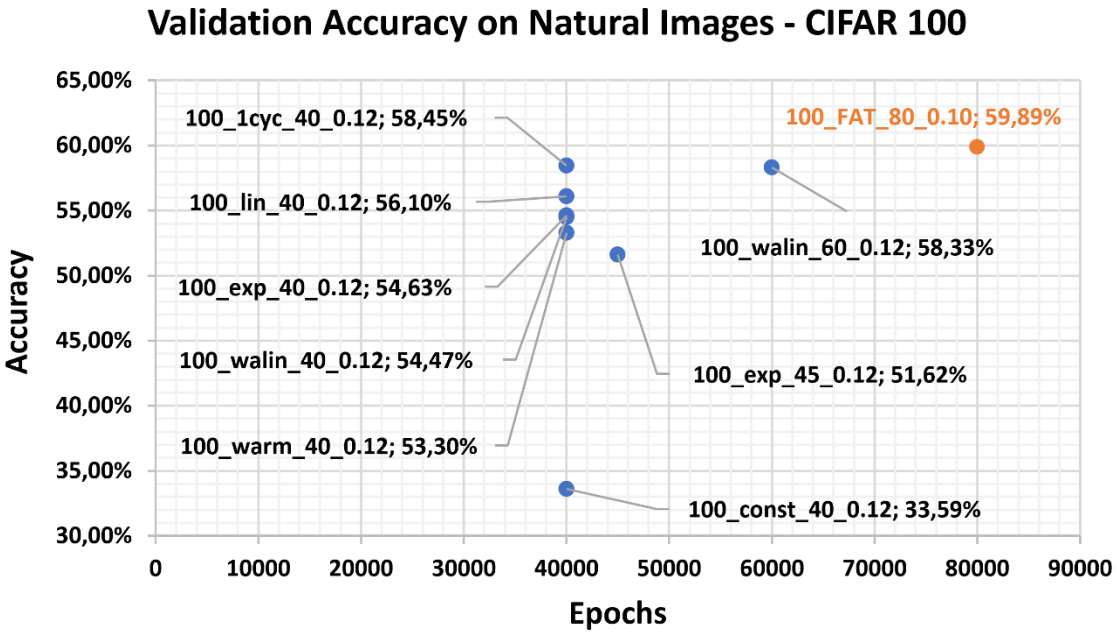

Figure 57 – CIFAR100 accuracy on natural images

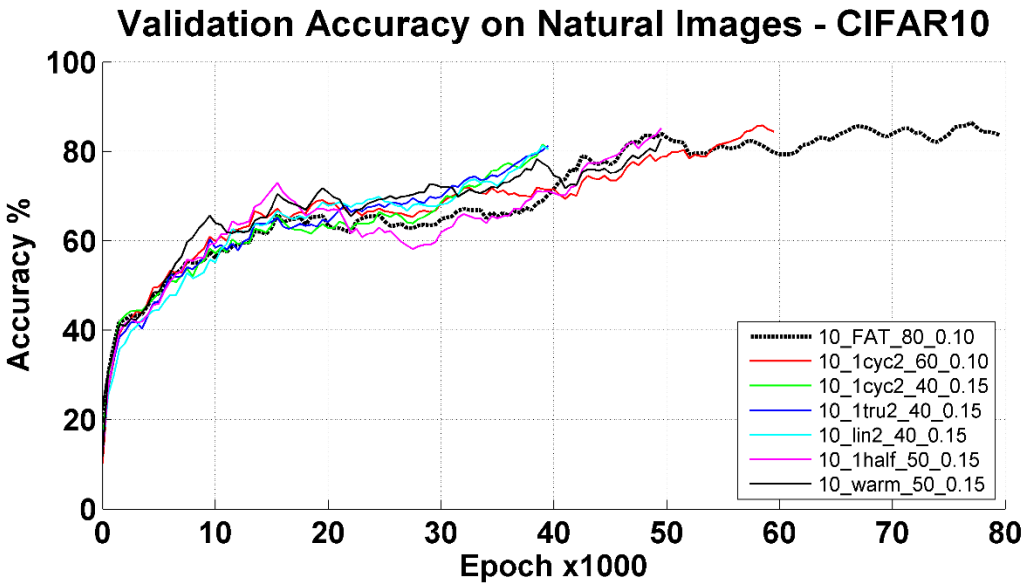

Figure 58 – CIFAR10 simulations (Part 1)

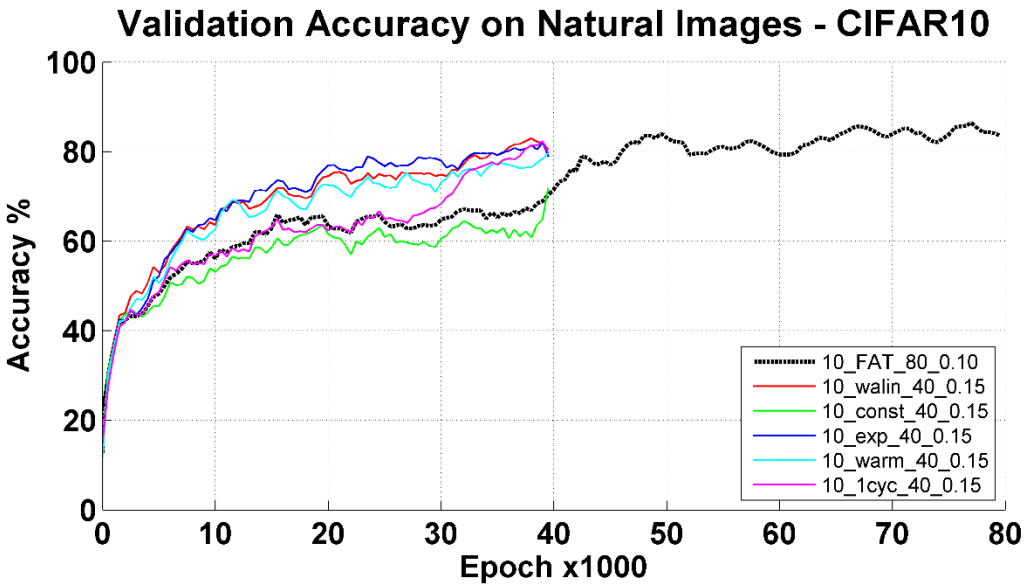

Figure 59 – CIFAR10 simulations (Part 2)

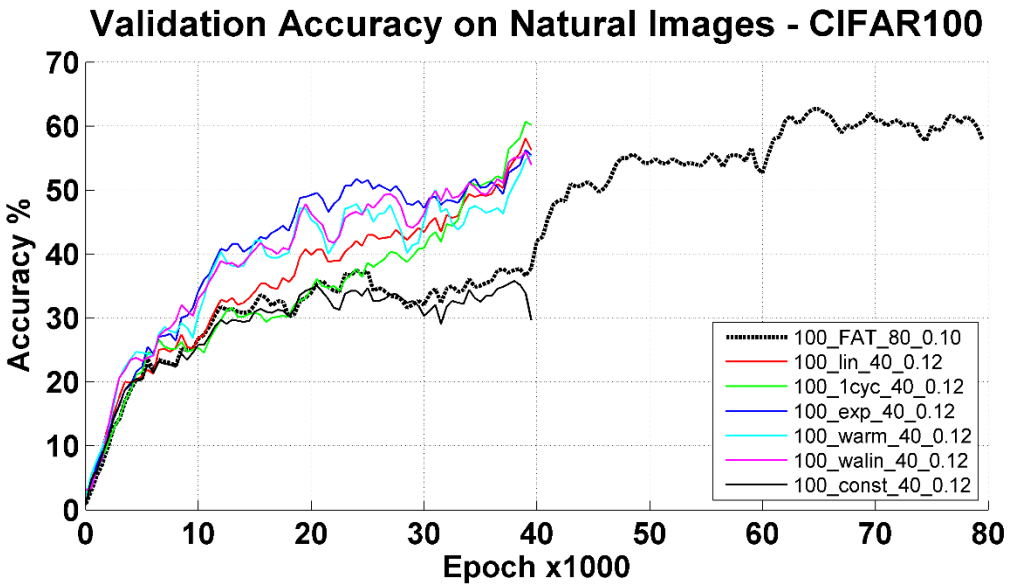

Figure 60 – CIFAR100 simulations

5.4.2 Adversarial images results

Figures 61 – 62 are the results obtained from the attacks. Each model suffered a PGD-20 attack, which is the same standard attack used against the original FAT.

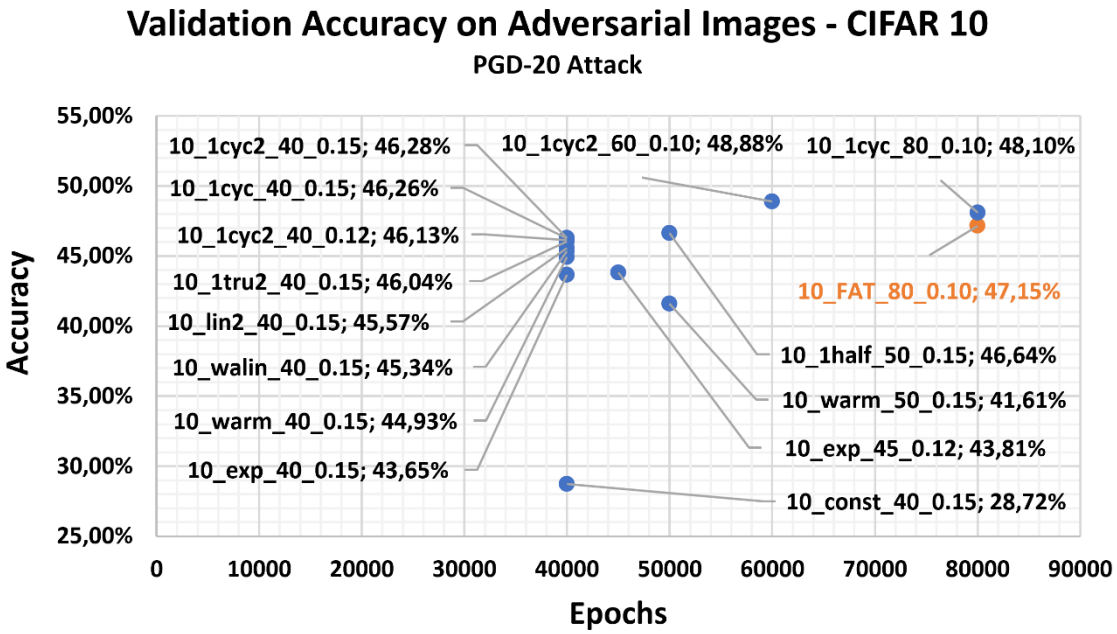

Figure 61 – CIFAR10 accuracy on adversarial images

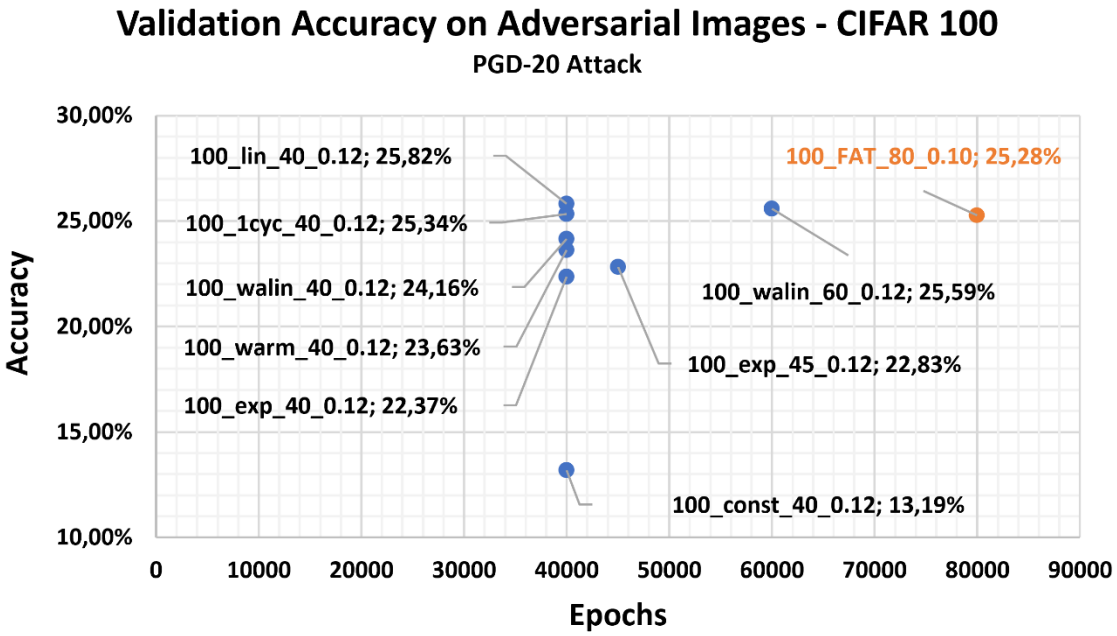

Figure 62 – CIFAR100 accuracy on adversarial images

5.5 Conclusion

As you can see in all the proposed graphs, the worst algorithm is the one with all constant hyperparameters, although the learning rate is maximum it cannot go deeper into the local minimum found. From these results it can be deduced that even the original FAT that has a 3-steps learning rate can be considered an optimization, but with more specific techniques the same results can be achieved even in half the time. The fine tuning of the hyperparameters is essential and can lead to super-convergence not only in normal training, but also for adversarial training, without affecting the result and robustness of the model itself.
The main method used for each simulation was a "trial and error" method with the aim of removing the plateau areas of the training, that is, when the accuracy does not increase for many epochs, but remains approximately constant. By removing the plateau areas, an ever increasing accuracy curve can be guaranteed, thus optimizing training times.

5.5.1 Future works

During the course of this thesis, new papers have been published both for attacks [65] and for fast adversarial training [66], so these are new starting points for

research.
Furthermore other techniques can be developed:

- Evaluation of the accuracy gradient to avoid plateau areas
- Partial image analysis
- Search for wide minimums for more robustness

All these ideas can be the basis for continuing research in this field.

Bibliography

[1] J. P. Mueller e L. Massaron, Machine Learning for dummies, For Dummies, 2016.

[2] S. Raschka e V. Mirjalili, Python Machine Learning, Packt Publishing, 2017.

[3] G. Toschi, «Il Machine Learning è divertente!,» 2017. [Online]. Available: https://medium.com/botsupply/il-machine-learning-%C3%A8-divertente-parte-1-97d4bce99a06.

[4] L. N. Smith, «A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay,» *https://arxiv.org/abs/1803.09820,* 2018.

[5] J. Brownlee, «Difference Between a Batch and an Epoch in a Neural Network,» 2018. [Online]. Available: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.

[6] Y. LeCun, C. Cortes e C. J. Burges, «THE MNIST DATABASE,» 1999. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[7] A. Krizhevsky, V. Nair e G. Hinton, «Cifar Datasets,» 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[8] F.-F. Li e C. Fellbaum, «ImageNet,» 2009. [Online]. Available: http://www.image-net.org/.

[9] D. Vasani, «How do pre-trained models work?,» 2019. [Online]. Available: https://towardsdatascience.com/how-do-pretrained-models-work-11fe2f64eaa2.

[10] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow e R. Fergus, «Intriguing properties of neural networks,» *https://arxiv.org/abs/1312.6199,* 2013.

[11] A. Marchisio, M. A. Hanif, S. Rehman, M. Martina e M. Shafique, «A Methodology for Automatic Selection of Activation Functions to Design Hybrid Deep Neural Networks,» *https://arxiv.org/abs/1811.03980,* 2018.

[12] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré e M. Zaharia, «DAWNBench: An End-to-End Deep Learning Benchmark and Competition».

[13] Y. Guo, C. Zhang, C. Zhang e Y. Chen, «Sparse DNNs with Improved Adversarial Robustness,» *https://arxiv.org/abs/1810.09619,* 2018.

[14] A. Katharopoulos e F. Fleuret, «Not All Samples Are Created Equal: Deep Learning with Importance Sampling,» *https://arxiv.org/abs/1803.00942,* 2018.

[15] J. Xu e S. Sala, «Guida alle architetture di reti profonde,» 2018. [Online]. Available: https://www.deeplearningitalia.com/guida-alle-architetture-di-reti-profonde/.

[16] A. Anwar, «Difference between AlexNet, VGGNet, ResNet and Inception,» 2019. [Online]. Available: https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96.

[17] V. Fung, «An Overview of ResNet and its Variants,» 2017. [Online]. Available: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035.

[18] K. He, X. Zhang, S. Ren e J. Sun, «Deep Residual Learning for Image Recognition,» *https://arxiv.org/abs/1512.03385,* 2015.

[19] S. Zagoruyko e N. Komodakis, «Wide Residual Networks,» *https://arxiv.org/abs/1605.07146,* 2017.

[20] H. Moore e M. Bassetti, Matlab per l'ingegneria, Pearson, 2008.

[21] M. Beri, Python, Apogeo, 2010.

[22] M. Buttu, Python, guida completa, Edizioni Lswr, 2014.

[23] gyanendra371, «Self in Python class,» 2019. [Online]. Available: https://www.geeksforgeeks.org/self-in-python-class/.

[24] M. Scarpino, TensorFlow for dummies, For Dummies, 2018.

[25] G. B. Team, «TensorFlow,» 2015. [Online]. Available: https://www.tensorflow.org/.

[26] L. V. Revision, «TensorFlow Installation,» 2018. [Online]. Available: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html#tensorflow-gpu.

[27] A. Sachan, «TensorFlow Model,» 2017. [Online]. Available: https://cv-tricks.com/tensorflow-tutorial/save-restore-tensorflow-models-quick-complete-tutorial/.

[28] D. Deutsch, «Debugging TensorFlow,» 2018. [Online]. Available: https://github.com/Createdd/Writing/blob/master/2018/articles/DebugTFBasics.md#5-use-the-tensorflow-debugger.

[29] N. Corporation, «CUDA,» 2007. [Online]. Available: https://developer.nvidia.com/cuda-zone.

[30] A. Salimath, «How to use the Learning Rate Finder in TensorFlow,» 2019. [Online]. Available: https://medium.com/octavian-ai/how-to-use-the-learning-rate-finder-in-tensorflow-126210de9489.

[31] A. Paszke, S. Gross, S. Chintala e G. Chanan, «PyTorch,» 2016. [Online]. Available: https://pytorch.org/.

[32] F. Chollet, «Keras,» 2015. [Online]. Available: https://keras.io/.

[33] S. Scardapane, «Breve guida agli adversarial examples nel machine learning,» 2019. [Online]. Available: https://iaml.it/blog/adversarial-examples?_waf=1#goodfellow2014.

[34] C. M. Wild, «Know Your Adversary: Understanding Adversarial Examples,» 2018. [Online]. Available: https://towardsdatascience.com/know-your-adversary-understanding-adversarial-examples-part-1-2-63af4c2f5830.

[35] A. Kurakin, I. Goodfellow e S. Bengio, «Adversarial examples in the physical world,» *https://arxiv.org/abs/1607.02533,* 2016.

[36] I. J. Goodfellow, J. Shlens e C. Szegedy, «Explaining and Harnessing Adversarial Examples,» *https://arxiv.org/abs/1412.6572,* 2015.

[37] J. Su, D. V. Vargas e S. Kouichi, «One pixel attack for fooling deep neural networks,» *https://arxiv.org/abs/1710.08864,* 2017.

[38] A. Madry, A. Makelov, L. Schmidt, D. Tsipras e A. Vladu, «Towards Deep Learning Models Resistant to Adversarial Attacks,» *https://arxiv.org/abs/1706.06083,* 2017.

[39] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik e A. Swami, «Practical Black-Box Attacks against Machine Learning,» *https://arxiv.org/abs/1602.02697,* 2016.

[40] S. Srisakaokul, Y. Zhang, Z. Zhong, W. Yang, T. Xie e B. Li, «MULDEF: Multi-model-based Defense Against Adversarial Examples for Neural Networks,» *https://arxiv.org/abs/1809.00065,* 2018.

[41] D. Wang, C. Li, S. Wen, S. Nepal e Y. Xiang, «Defending against Adversarial Attack towards Deep Neural Networks via Collaborative Multi-task Training,» *https://arxiv.org/abs/1803.05123,* 2018.

[42] L. Schmidt, S. Santurkar, D. Tsipras, K. Talwar e A. Madry, «Adversarially Robust Generalization Requires More Data,» *https://arxiv.org/abs/1804.11285,* 2018.

[43] Y. Esfandiari, K. Ebrahimi, A. Balu, N. Elia, U. Vaidya e S. Sarkar, «A Saddle-Point Dynamical System Approach for Robust Deep Learning,» *https://arxiv.org/abs/1910.08623,* 2019.

[44] A. Shafahi, P. Saadatpanah, C. Zhu, A. Ghiasi, C. Studer, D. Jacobs e T. Goldstein, «Adversarially robust transfer learning,» *https://arxiv.org/abs/1905.08232,* 2019.

[45] S. Gui, H. Wang, C. Yu, H. Yang, Z. Wang e J. Liu, «Adversarially Trained Model Compression: When Robustness Meets Efficiency,» *https://arxiv.org/abs/1902.03538,* 2019.

[46] N. Papernot, P. McDaniel, X. Wu, S. Jha e A. Swami, «Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks,» *https://arxiv.org/abs/1511.04508,* 2015.

[47] J. H. Metzen, T. Genewein, V. Fischer e B. Bischoff, «On Detecting Adversarial Perturbations,» *https://arxiv.org/abs/1702.04267,* 2017.

[48] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi e P. Frossard, «Universal adversarial perturbations,» *https://arxiv.org/abs/1610.08401,* 2016.

[49] J. Rauber e W. Brendel, «Foolbox,» 2017. [Online]. Available: https://foolbox.readthedocs.io/en/stable/index.html#.

[50] I. Goodfellow e N. Papernot, «Cleverhans,» 2016. [Online]. Available: http://www.cleverhans.io/.

[51] J. Rauber, W. Brendel e M. Bethge, «Foolbox: A Python toolbox to benchmark the robustness of machine learning models,» *https://arxiv.org/abs/1707.04131,* 2017.

[52] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik e A. Swami, «The Limitations of Deep Learning in Adversarial Settings,» *https://arxiv.org/abs/1511.07528,* 2015.

[53] H. Zhang, Y. Yu, J. Jiao, E. P. Xing, L. E. Ghaoui e M. I. Jordan, «Theoretically Principled Trade-off between Robustness and Accuracy,» *https://arxiv.org/abs/1901.08573,* 2019.

[54] D. Zhang, T. Zhang, Y. Lu, Z. Zhu e B. Dong, «You Only Propagate Once: Accelerating Adversarial Training via Maximal Principle,» *1905.00877,* 2019.

[55] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. S. Davis, G. Taylor e T. Goldstein, «Adversarial Training for Free!,» *https://arxiv.org/abs/1904.12843,* 2019.

[56] T. Lienart, «Projected gradient descent,» 2018. [Online]. Available: https://tlienart.github.io/pub/csml/cvxopt/pgd.html.

[57] S. Gugger e J. Howard, «AdamW and Super-convergence is now the fastest way to train neural nets,» 2018. [Online]. Available: https://www.fast.ai/2018/07/02/adam-weight-decay/.

[58] L. N. Smith e N. Topin, «Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates,» *https://arxiv.org/abs/1708.07120,* 2018.

[59] H. Song, S. Kim, M. Kim e J.-G. Lee, «Ada-Boundary: Accelerating the DNN Training via Adaptive Boundary Batch Selection,» *https://openreview.net/forum?id=SyfXKoRqFQ,* 2018.

[60] S. Gugger, «How Do You Find A Good Learning Rate,» 2018. [Online]. Available: https://sgugger.github.io/how-do-you-find-a-good-learning-rate.html.

[61] S. Gugger, «The 1cycle policy,» 2018. [Online]. Available: https://sgugger.github.io/the-1cycle-policy.html.

[62] L. N. Smith, «Cyclical Learning Rates for Training Neural Networks,» *https://arxiv.org/abs/1506.01186,* 2017.

[63] I. Loshchilov e F. Hutter, «SGDR: Stochastic Gradient Descent with Warm Restarts,» *https://arxiv.org/abs/1608.03983,* 2017.

[64] P. Mishra e K. Sarawadekar, «Polynomial Learning Rate Policy with Warm Restart for Deep Neural Network,» *https://ieeexplore.ieee.org/document/8929465,* 2019.

[65] D. Goodman, H. Xin, W. Yang, X. Junfeng, Z. Huan e W. Yuesheng, «Advbox: a toolbox to generate adversarial examples that fool neural networks,» *https://arxiv.org/abs/2001.05574,* 2020.

[66] E. Wong, L. Rice e J. Z. Kolter, «Fast is better than free: Revisiting adversarial training,» *https://arxiv.org/abs/2001.03994,* 2020.