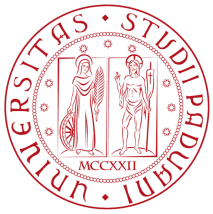


UNIVERSITY OF PADUA

Department of Information Engineering

Master's Degree in Computer Engineering

Final Dissertation

Robustness Analysis of Self-Supervised Models

Supervisor:

Prof. Gian Antonio Susto

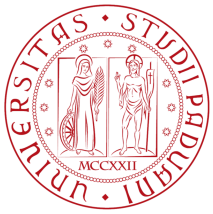
Co-Supervisor:

Dr. Matteo Terzi

Candidate:

Federico Sergio

Academic Year 2021-2022



UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Analisi sulla Robustezza di Modelli Self-Supervised

Relatore:

Prof. Gian Antonio Susto

Correlatori:

Dott. Matteo Terzi

Laureando:

Federico Sergio

Anno Accademico 2021-2022

Aknowledgments

I will always be grateful to my family for the support they have given to me throughout my university career and especially during the writing of this thesis. I also want to thank Andrea, Claudio and all the other special friends who have spurred me through this long and difficult journey. A special mention goes also to Matteo, who had the great patience to guide me in the elaboration of this work.

Padua, 28 February 2022

Abstract

In today's society, the use of Machine Learning technologies is increasingly supporting human activities, from research contexts to work, up to aspects of daily life. If from one hand this phenomenon has allowed the resolution of individual problems at a speed never seen before, on the other hand, it is necessary to consider that in several fields of application the reliability of these tools is still far from safe use. The introduction of learning techniques that are less and less subject to human intervention, such as the review and data labeling, also focuses attention on the implementation of these new algorithms, to improve their general performance.

In this thesis work, we intend to deepen the aspects introduced with the implementation of an algorithm of Deep Learning based on the Self-Supervised technique, to perform an analysis of the crucial components in model learning and reinforcement process. This algorithm is designed with an Adversarial Training module, to evaluate the model and to investigate its costs and benefits in terms of robustness.

Sommario

Nella società odierna l'impiego delle tecnologie di Apprendimento Automatico affiancano sempre di più le attività di impiego umane, dai contesti di ricerca all'utilizzo lavorativo, fino ad aspetti di vita quotidiana. Se da una parte questo fenomeno ha permesso la risoluzione di singoli problemi ad una velocità mai vista prima, dall'altra è doveroso considerare che in diversi campi di applicazione l'affidabilità di questi strumenti è ancora lontana al fine di un utilizzo in sicurezza. L'introduzione di tecniche di apprendimento sempre meno soggette ad un intervento umano, come la revisione e l'etichettamento dei dati, focalizza inoltre l'attenzione sul funzionamento dei nuovi algoritmi, al fine di migliorarne le prestazioni generali.

In questo lavoro di tesi si intendono approfondire gli aspetti introdotti con l'implementazione di un algoritmo di *Deep Learning* basato sulla tecnica *Self-Supervised*, al fine di effettuare un'analisi delle componenti determinanti nel processo di apprendimento e rinforzo del modello. In tale algoritmo viene infatti implementato un modulo per fare *Adversarial Training*, così da poter valutare correttamente il modello ed approfondirne costi e benefici in termini di robustezza.

Contents

Chapter 1

Robustness of Deep Learning Models

This Chapter is intended to give to the reader a brief explanation of the main topics behind this thesis. Section 1.1 explains Adversarial principles and the basics of Self-Supervision. Section 1.2 serves as headlight before Chapter 5, to simply illustrate the overall focus in the research and how the working process was built from the beginning to the end.

1.1 Introduction to the topic

1.1.1 Adversarial Examples

State of the art in deep learning allows the application of numerous algorithms in a large variety of fields, in which a strict level of precision on the unfolding tasks is requested, to guarantee safe and effective usage. *Computer Vision* applications, like self-driving cars and *Face Recognition*, or *Natural Language Processing* application, such as malware detection, are just a few implementation fields that require a high level of reliability. For this reason, the need for an analysis of the critical and security aspects of the classifiers arises.

In the contexts presented previously is a wide object of study the sphere of the so-called *adversarial examples*, meaning those data that generate incorrect output despite a high classification accuracy of the model, both for intrinsic characteristics, both for their manipulation. The existence of this phenomenon suggests a flaw, not in the data itself (the goal is being able to predict theoretically all kinds of data in the domain of interest) but rather it's a sign of feature generalization lack in the model. Hence the need to develop a so-called robust model, i.e. a model that can effectively classify even the most intrinsically complex inputs.

Speculative explanations in the past suggested adversarial examples exist due to the extreme non-linearity of deep neural networks, combined with models overfitting [54]. Following studies proved that previous assertions were not correct. In fact what makes the difference against adversarial attacks is a change of nonlinear model families, with a cost in terms of training simplicity. It is thus more convenient to design more powerful optimization methods that can successfully train more nonlinear models than to focus on the regularization of linear ones [16]. To summarize, adversarial examples can be explained as a property of high-dimensional dot products. They are a result of models being too linear, rather than too nonlinear.

While the great advantage of Deep Learning lies in the ability to obtain good representations of data, it has been shown that classifiers do not use the same human criteria when training and predicting tasks [34]. Rather, in the specific case of adversarial samples, it was discovered that these deficiencies in generating an effective representation are due to elements that are often indistinguishable from the human eye. There are two main ways to describe this kind of input:

1. Natural real-life examples, where tracing misclassification reasons on an already trained algorithm is not possible.

2. Pre-given data with even a slightly, man-made alteration in its information, in such a way that those changes can deceive the algorithm and make it substantially unreliable [12].

These definitions are somehow correlated. The first one describes how adversarial examples occur in reality, the second explains the process of training robust classifiers, forcing the algorithm to inspect hard samples generated from the set itself. One could sustain that this process is essentially the same as *data augmentation*. Training on adversarial examples is somewhat different from other data augmentation schemes; usually, one augments the data with transformations such as translations or rotations, that is expected to actually occur in the test set. This form of data augmentation instead uses inputs that are unlikely to occur naturally but that expose flaws in the ways that the model conceptualizes its decision function [16].

Introduced the basic principles of the Adversarial phenomenon, in the next section we explain what kind of model we will submit to Adversarial inputs.

1.1.2 Reasons Behind Self-Supervision

In this thesis work, we wanted to integrate the aspects mentioned above with an accurate study of the model used for the experimental phase. Particular attention has been paid to Self-Supervised Learning (SSL). These models represent a valid solution in fields such as speech and image recognition. The Self-Supervised Frameworks do not require labels for training to formulate predefined tasks. This aspect makes SSL models less prone to the problem of data labeling, which is instead necessary for Supervised Training and requires human intervention to manually label individual inputs. In a real context to create an accurate predictor on the desired domain, Deep Neural Networks need to process datasets that are thousands, even millions in sample size. Despite the fact we live in a big data era, still, the cost of high-quality human labeling could be highly expensive, as Scale.ai[1] (data labeling company) witnesses. The scale aspect when choosing the model type is thus crucial once we focus on efficiency.

It is quite easy to confuse Self-Supervised Learning and Unsupervised Learning (UL). In some way, SSL can be seen as a sub-branch of UL considering the same no-label context. Although the Unsupervised approach is mainly aimed at finding patterns in the data, such as clustering or anomaly detection, self-supervision concentrates on recovering, a typical supervised behavior [46].

As Chapter 5 explains, typically SSL models are designed to make predictions with the application of *data augmentation* before providing the input to the model, for example, to find the context of a sentence or the rotation of an image [60, 29]. As it will be deepened also in Chapter 3, the choice of the transformations applied to the data turns out crucial in this field, helping to increase the consistency and the quality of the model [52]. Moreover, it has been widely demonstrated how, using different types of datasets and Neural Network architectures, Self-Supervised Learning has achieved even better results than other Supervised Learning techniques [7].
Thus, the Self Supervised approach turns out to be, in the specific fields of application, a good compromise between reliability and scalability of classifiers.

[1] `https://scale.com/pricing`

1.2 Main Goals

Self-Supervised Learning and Adversarial Training will be the main objects of investigation in this research. The initial task is to combine the two machine learning approaches into a single code: adapting the generation of adversarial samples according to the model used and collecting data of its behavior based on multiple parameters and databases. In order to implement SSL, nowadays there are several implementation techniques, based on different conceptual implementations that will be widely covered in Section 3.3. In the examination case, SimCLR will be used [7], which is a framework of Contrastive Learning that affects predictions based on the "similarity" of more inputs. This will also allow us to apply Adversarial Training in a very specific context, which can be used to further investigate how the model learns features. To deeply understand how well the model generalizes data, it will then be necessary to introduce additional elements for feature evaluation, such as fine-tuning the previously trained models, using natural and adversarial data, to observe the results thus obtained. The final goal is multiple, combining different learning techniques it is indeed possible to make several types of analysis:

- Observe how the algorithm generates semantically adversarial samples.
- Identify what features are used by the model during both the contrastive and standard classification process.
- Understand whether, in examination cases domain, Adversarial Training actually helps improve model robustness in multiple scenarios:
  - When a trained model is used to solve the same identical task.
  - When a trained model is subjected to tasks other than the starting one (like a task in a similar but more complex domain compared with the first).
- Observe the behavior of the SSL model when only certain training steps are targeted in Adversarial Training process.

After this brief overview of the covered topics, we can proceed to a more accurate understanding of the technical aspects that are part of this research activity. In the next chapter, an introduction of Deep Learning theory is presented, with a specific eye on the notions useful to better understand the test phase of this elaboration.

Chapter 2

Deep Learning Overview

With the term *Deep Learning* (DL) we refer to a set of techniques, structures and algorithms in the domain of Artificial Intelligence (AI), specifically part of *Machine Learning* (ML) domain. Machine learning differs from classical AI approaches, in ML human does not provide explicit instructions to the machine, with a case by case analysis to make predictions from a set of prior knowledge (i.e, data), but designs the architecture that allows the latter to make inference directly from the input. The relationship between the different aspects introduced is illustrated with the Venn diagram in Figure 2.1. The limitations of conventional algorithms represented an incentive in DL research, strictly related to an analysis of the "raw" data available [10].

Deep Learning algorithms can be defined as multi-layer representation techniques (i.e., based on a multiple *layer* structure) that allow one to generate a high-level abstract representation from a set of simple input data. This is made possible by the use of several nonlinear modules that transform the inputs during execution [41]. Due to this peculiarity, the field of Deep Learning has been associated in recent years with *representation learning*, where hierarchies of *features* are used so that higher levels are derived from lower ones, and the latter in turn help to define the former. These models are a mathematical representation of the relationship between inputs and outputs, where the learning process is to estimate as accurately as possible the parameters that the model must set to solve a given *task* [61].

The architectures commonly used in DL are the **Deep Neural Networks (DNN)** or Artificial Neural Networks. Their operation is inspired by the structure of the human brain, composed of several connections between a multitude of neurons that, when stimulated with certain signals, are activated and propagate information to their successors in the network. The technical functioning of the brain is not the only source of inspiration for DNNs, but conceptually it is the natural demonstration that we learn from experience. Problems that are

beyond the capabilities of today's computers can in fact be solved with many small efficient "modules" that avoid extremely technical solutions at the machine level [35]. A neural network is much more than a group of neurons connected. What makes the difference is the *layering*: this aspect is reproduced in DNNs, which are divided into multiple groups connected according to specific criteria, each of which represents a different stage in the learning process. The artificial neural network is conceptually a graph, where the computational elements are located in the single nodes, while the weights are located in correspondence of the arcs.

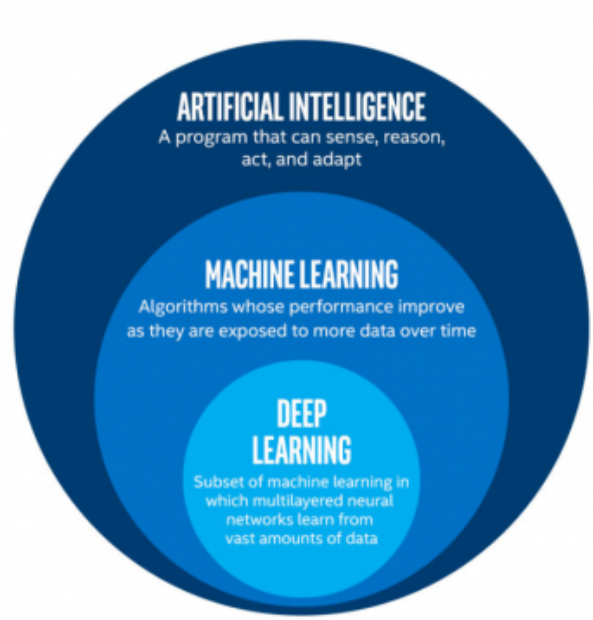


Figure 2.1: Hierarchy of Fields of Interest. (1) Artificial Intelligence: a program that can perceive, reason, act, and adapt. (2) Machine Learning: algorithms whose accuracy increases with the amount of data provided. (3) Deep Learning: multi-layered neural networks that learn from massive amounts of data [26].

2.1 Core Elements of Deep Neural Networks

2.1.1 Historical Introduction

Although Deep Learning has been widely known commercially for about 20 years, the first studies date back to the 1940s, when McCulloch and Pitts published the first study focused on the transposition of neuron's functioning on a mathematical model, in 1943 [36]. However, such a study represents a

first, primitive attempt to "map" the human brain. The first widely described model of a neuron belongs to Frank Rosenblatt, who gave in 1958 the first definition of *Perceptron* [45]. Rosenblatt made a great contribution to the scientific community but did not formulate how a single module should work in conjunction with the others.

It was not until 1965 that the first supervised deep feedforward multilayer perceptrons were published (although such a definition did not yet exist). The study, conducted by Invakhnenko and Lapa, introduced units with polynomial activation functions combined with addition and multiplication operations [24]. The same Ivakhnenko in 1971 described a deep network with 8 layers trained with the "Group Method of Data Handling" [25], then maintained by the scientific community in the following decades. This method consists of training incrementally the layers with regression analysis, starting from a training set of inputs with their respective outputs. By using regularization, the superfluous units were discarded, this was possible since it was also used a validation set. In summary, Ivakhnenko found a way to describe a network that would generate an internal representation of incoming data through a hierarchical and distributed structure.

Having laid the foundations for the deepening of DL, in the following years' new methods were born to analyze the input data. With Fukushima's study (1979-1982) concerning visual pattern recognition, a new space was opened in the framework of machine learning architectures [13]. Under the name of "Neocognitron", the first prototype of Convolutional Neural Network (CNN) was defined, whose aspects will be extensively discussed in Chapter **??**. The model implements a rectangular receptive field with a vector of weights (essentially a filter) that shifts along with a two-dimensional array of input values (in the practical case, image pixels). This process, iterated along all the networks, generates a massive number of weights describing the various convolutional layers, feeding downsampling layers composed of fixed-weight connections originating from layers below. The network implements the downsampling units using the *Spatial Averaging* technique, which activates it only if at least one of the inputs is active. Later this technique was replaced by the *Max-Pooling*.

An important year in the development of machine learning was 1989, when the standard backpropagation algorithm was formulated by Yann Le Cun [31]. The algorithm was applied to recognize handwritten zipper code digits. Basically, it was a reverse version of the *automatic differentiation* method[1]. How-

[1]Mathematical method applied in Neural Networks consists in computing the derivatives of the output error concerning each weight. By 1980, automatic differentiation could derive Backpropagation for any differentiable graph [53].

ever, the execution timing was not yet such as to make it easily applicable on the systems of that time. Le Cun's work was also based on the implementation of Recurrent Neural Networks (RNNs), later also trated by Schuster and Paliwal in 1997[47]: these can recreate for/while loops and also do recursion, besides learning to solve problems of potentially unlimited depth. In general, however, finding an NN that precisely models a given training set (of input patterns and corresponding labels) is an NP-complete problem, also in the case of deep NNs.

All the studies conducted until the beginning of the new millennium brought several contributions to the scientific community in the field of machine learning and specifically Deep Learning, but many were the theoretical and practical complications that did not allow this discipline to become popular in computer science. Practical flaws such as the time limit of execution of certain algorithms or the *vanishing gradient* problem described in more detail in Section 2.2, contributed to discard the NNs as a potential field of study. Only in 2006 DL was universally recognized for its enormous potential, when the study of Geoffrey E. Hinton, Simon Osindero and Yee-Whye Teh demonstrated how to eliminate the difficulties encountered previously using a new, fine-tuned, generative model, able to perform digit classification better than the best discriminative learning algorithms [21]. In addition, this model could visualize what the associative memory had in mind.

2.1.2 Feedforward Artificial Neural Networks

To better understand how the whole Deep Learning process works, we introduce how an Artificial Neural Network is built. Using a "bottom-up" approach this Section first introduces the single neuron unit, i.e. Perceptron algorithm. Secondly, we give an overall view of the Feedforward Neural Networks (FNNs), as combinations of several neurons, also known as Multi-Layer Perceptrons (MLPs).

2.1.2.1 Perceptron

As we have seen in Section 2.1, the first mathematical model allowing to define a primordial functioning of the Artificial Neural Networks was the Perceptron [45]. We can in fact define this model as a single-layer neural network, which describes (as a unit) a binary classifier. The model is composed of 4 main sublayers:

1. An $n + 1$ dimensional vector of *inputs* $\boldsymbol{x}$ which contain the information to be processed.

2. An n dimensional vector of *weights* $\mathbf{W}$, initially instantiated to 0 or in a random fashion, plus a *bias* constant b.

3. A so-called *Net Sum*, i.e the weighted sum unit of the input vector with the weight vector.

4. An *activation function*[2] assigned to map the result of the previous sum into a range (usually (0, 1) or (-1, 1)). Then, depending on the result, this function indicates whether the perceptron will be activated, or in other words whether the result is returned as output.

A visual representation of a Perceptron is shown in Figure 2.2. One powerful aspect of this model lies in the input format. Indeed the model can in practice be fed with inputs of different algebraic formats, and thus can work with several types of data. It will be more clear what this means in practice in Section **??**, describing how images are given to each unit.

The weight vector represents the core of the model, because it is the numerical representation of how the model is generalizing the function outlined by our set of inputs. The more precisely the weights describe the generalization function, the better classification will be performed. Moreover, bias is a fundamental component of a Perceptron, since it shifts the activation function curve up or down, enhancing the final choice of the unit [41]. All of these computations result in the output vector y with the following form:

$$y = f_A(\mathbf{W}^T\boldsymbol{x} + b) = f_A(w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + ... + w_n\boldsymbol{x}_n + b) \tag{2.1}$$

where w_i represents the i-th element of the weights vector, x_i the i-th element of the input vector and $f_A : \mathbb{R} \to \mathbb{R}$ the activation function.

By itself, a perceptron alone is not a versatile classifier, because it can learn only linearly separable problems. For instance, consider the function OR in a binary domain: in this case, our model will be able to find a linear function to classify the outputs, since such a function exists (in the described domain, the line passing through the points (0, 1) and (1, 0)). However, this does not apply to other basic operations such as $\boldsymbol{x}$ OR or NXOR, which require a non-linear function to be mapped. To reach a higher level of generalization, one needs to

[2]We will better cover the technical aspects of activation functions in Section 2.1.3

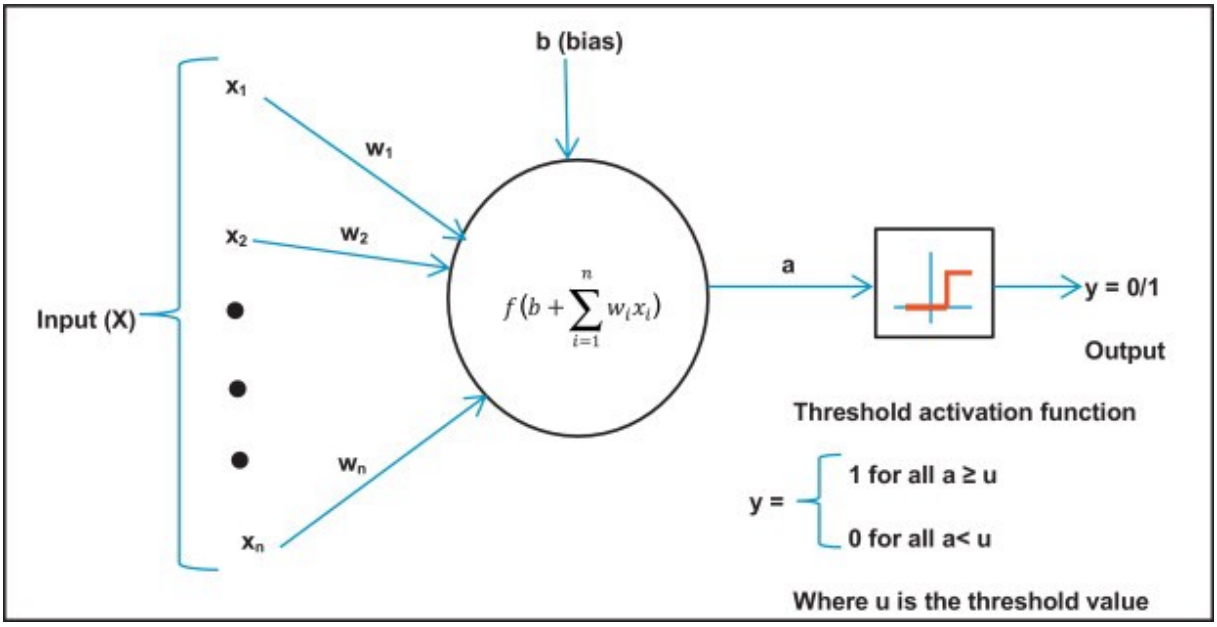


Figure 2.2: Perceptron Structure. The bias is added after the algebraic sum of the elements of the input vector $\boldsymbol{x}$, weighted with those of the vector $\mathbf{W}$. The result is passed (in the figure) to an activation function *Heaviside*, also called "Step function", which determines if the information can be released as an output vector y. [32]

put more of these units together, so that a more complex and flexible structure can be used to resolve tasks.

2.1.2.2 Multilayer Perceptron

A *Multi-Layer Perceptron (MLP)* is an ML architecture composed of 3 distinct sets of layers. Each layer implements a number of perceptrons based on data format, the amount of complexity and generalization to be obtained, and the output to analyze. The first is the *input layer*, which begins the computation. The last layer, called the *output layer* performs tasks such as predictions and classification [1]. The second set is called *hidden layer*, this is the most important part of the structure since what happens here can not be controlled by the programmer directly, but happens automatically based only on input parameters and regularization techniques that we will see in Section 2.1.4. Moreover, the number of hidden layers is based on engineering choices and indicates the *depth* of the network, while the one containing the largest amount of neurons determines its *width*.

In MLPs data flows forward from input to output layer. They are designed

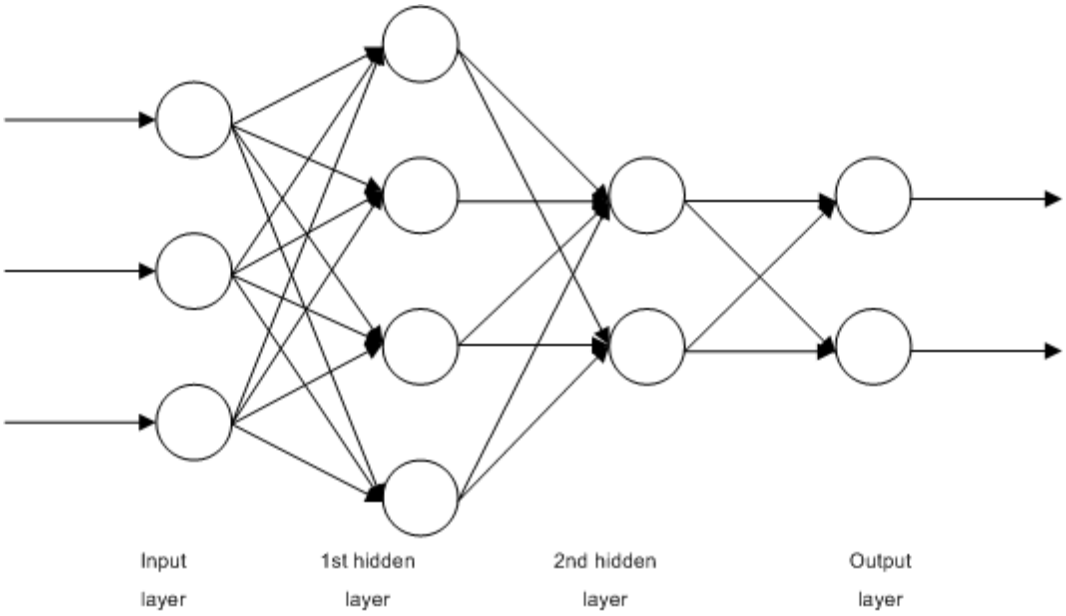


Figure 2.3: Multilayer Perceptron. [37]

to approximate any continuous function and can solve non-linear problems (i.e., which are not linearly separable). A representation of MLPs is given by 2.3.

Usually, we refer to DL when the task to be solved implies the use of a deep neural network (DNN) $\mathcal{N}$ with more than one hidden layer, so when its depth $d_{\mathcal{N}} > 1$. In order to make DNNs properly useful, non-linearity of activations functions in hidden layers is essential. The reason behind lies in the fact that every single unit's weighted sum of inputs (also shown in (2.1)) is an affine transformation. If non-linearity is not introduced into the model, then using more than one layer would be useless since a combination of linear functions results in a new linear function. Thus it means, more than one hidden layer can be always reduced mathematically into a single one, foreclosing to explore a large portion of real problems. Let x_{i-1} be the input of a single neuron in the i-th hidden layer, $\mathbf{W}_i$ the weight matrix of hidden layer i and b_i the respective bias, then computations taking place in hidden layers are as follows:

$$h_i(x_{i-1}) = f_A(\mathbf{W}_i \boldsymbol{x}_{i-1} + b_i), 0 < i < d \tag{2.2}$$

with d to be the depth of a DNN $\mathcal{N}$, $f_A : \mathbb{R} \to \mathbb{R}$ and $g_A : \mathbb{R} \to \mathbb{R}$ two non-linear activation functions, then the output results in:

$$o(x_{d-1}) = g_A(\mathbf{W}_d h(x_{d-1}) + b_d) \tag{2.3}$$

The main purpose of a DNN is to learn the best set $\vartheta = \{\mathbf{W}_i, \mathbf{W}_d, \mathbf{b}_i, \mathbf{b}_d\}$ such that the expectation of the *loss function* on relative task is minimized. A deeper explanation about these aspects is approached in Section 2.1.4.

2.1.3 Activation Functions

As thoroughly explained in the previous section, the nonlinearity of linear functions plays a key role in the proper functioning of DNNs. Making each function independent from the others is certainly a key aspect, but not the only one. The use of such functions also serves to give a predetermined meaning to the data coming out of each unit. When each output has the same fundamental numerical properties, it is possible to relate them to obtain information that leads to generalization. Since individual weights and biases can have very different values in complex networks, the activation functions also serve as a normalizer to standardize the processed data.

One feature to consider when approaching AFs, therefore, is the output range, which establishes in what neighborhood the result will necessarily be found. Each function is better suited for certain tasks rather than others, in other words, some AF are more useful in hidden layers while others, more suitable for classification, in the output layer. Specifically, we can group the AFs into two macro-groups, that of functions that simplify the signal, without drastically limiting it, and that of functions that reduce the input to an interval. Below we present different types of AF to better understand how they are used in practice.

2.1.3.1 The "Intermediate" Activations

We define the group of the following functions as "Intermediate" since usually these classes are used in the middle of the computational process. These AFs do not distort the input but simplify it by considering only a part of it.

When in 2010 Nair and Hinton first proposed the *Rectified Linear Unit (ReLU)*[3] function, it quickly became a standard in DNNs architectures [40]. In the following years were created different variants, each with its own peculiarities. Because of their features they are used for standard computation in hidden units of the network, which do not have to perform special classifications.

[3]ReLU function is also used by the ResNet architecture, a neural network with which the experimentation phase has been conducted

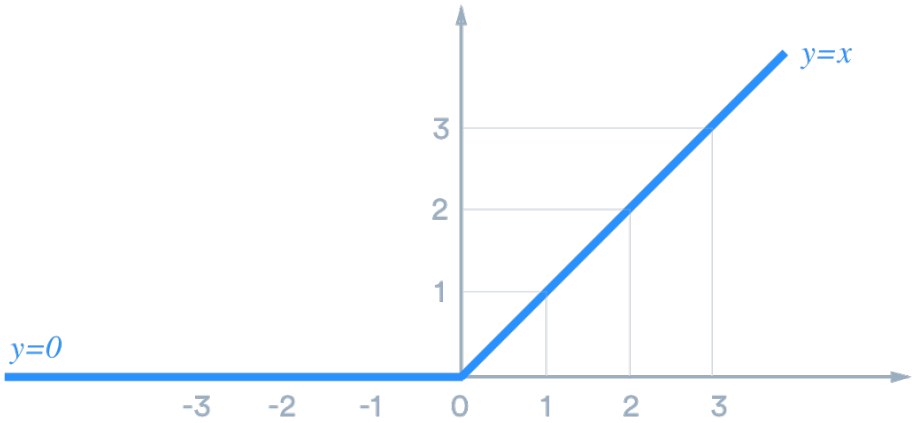


Figure 2.4: Rectified Linear Unit function. Negative abscissae are conform to zero ($y = 0$), while positive abscissae follow the trend of the input ($y = \boldsymbol{x}$). [56]

- **ReLU**: Among the various commonly used AFs is the most popular since it demonstrates better performance and greater generalization in DL than others The ReLU is an "almost-linear" function, which unlike other AFs turns out to preserve the properties of linear models, a feature that makes the model easy to optimize by methods of *gradient descend* [14]. It is a threshold function, which sends negative inputs to 0 while keeping positive inputs unchanged. Given the input vector $\boldsymbol{x}$, ReLU can be defined as follows:

$$ReLU(\boldsymbol{x}) = max(0, \boldsymbol{x}) = \begin{cases} \boldsymbol{x}_i & \text{if } x_i \geq 0 \\ 0 & \text{if } x_i < 0 \end{cases} \tag{2.4}$$

  The main advantage of using ReLU lies in computational simplicity, as it does not use exponentials or divisions, which in the context of DNNs translates into higher execution speed. Among usual AFs, computing *max* function is in fact way simpler than previously mentioned operations. Moreover, it avoids vanishing gradient problems, i.e. prevents the gradient of the network to become increasingly smaller, which could lead the algorithm to undesired results. More informations about vanishing gradient are described in Section 2.2.1.2. However, ReLU is more easily subject to *overfitting* than the Sigmoid, a problem that can be solved with the *Dropout* technique, which preempts the synchronous optimization of all the weights of the network (for details on how dropout works, see Section 2.2.2.2).

- **Leaky ReLU**: When it comes to using ReLU, the negative part is completely nullified. Instead, we could maintain a diminished part of it creating the following function:

$$LeakyReLU(\boldsymbol{x}) = max(0, \boldsymbol{x}) + min(\lambda \cdot \boldsymbol{x}, 0), \tag{2.5}$$

  where λ is a *leak factor* which serves to include also a small negative part in the equation. Also, allowing at least some part of the signal to go through, reduces the vanishing gradient problem, which we said is more probable for standard ReLU instead.

- **Exponential Linear Unit**: This variant is based on the same concept as *leaky*, but as the name suggests, the exponential function for the negative part is introduced. In this way, only a small portion of the negative input is retained, since the exponential converges to 0 on the abscissae in that direction.

$$ELU(\boldsymbol{x}) = max(0, \boldsymbol{x}) + min(e^{\boldsymbol{x}-1}, 0). \tag{2.6}$$

  In practice, ELU performs better than *leaky* ReLU when compared with standard ReLU. Furthermore, the use of this function in recent years has led to its refinement, there exists in fact the *Scaled Exponential Linear Unit (SELU)*, which is an activation function that allows to self-normalize the input signal. This translates into an output with zero mean and unit variance, which actually nullifies batch normalization layers. Moreover, positive and negative values are scaled using proper constants as follows:

$$SELU(\boldsymbol{x}) \approx 1.0507 \cdot max(0, \boldsymbol{x}) + 1.7580 \cdot min(e^{\boldsymbol{x}} - 1, 0) \tag{2.7}$$

  Further details on the Exponential Unit and the origin of SELU constants can be found at [27].

2.1.3.2 The "Final" Activations

Usually, these types of functions are used in the last layer of the network since the output is always in a finite range, and it is, therefore, possible to perform feature extraction and classification operations.

- **Tanh**: Is the *Hyperbolic Tangent* function. It is an S-shape function with an output range of $[-1, 1]$. It is clear that the output of Tanh function is zero centered, in this way we can easily map the output as strongly positive, neutral, or negative with a uniform magnitude.

  The function has the following form, for any $\boldsymbol{x} \in \mathbb{R}$:

$$Tanh(\boldsymbol{x}) = \frac{(e^{\boldsymbol{x}} - e^{-\boldsymbol{x}})}{(e^{\boldsymbol{x}} + e^{-\boldsymbol{x}})} \tag{2.8}$$

- **Sigmoid**: This function takes any real input $\boldsymbol{x} \in \mathbb{R}$ and maps it into the range $[0, 1]$. Because of its range, Sigmoid happens to be a good choice when it comes to translating data in a probability value, then easily convertible or interpreted in percentage. It has a similar shape compared to Tanh and it is computed as follows:

$$Sigmoid(\boldsymbol{x}) = \frac{1}{(1 + e^{-\boldsymbol{x}})} \tag{2.9}$$

- **Softmax**: Commonly used in classification problems, it is a generalization of a logistic function that compresses a k-dimensional vector $\mathbf{z}$ of arbitrary real values into a k-dimensional vector $\sigma(\mathbf{z})$ of values within an interval (0, 1) whose sum is 1. The function appears in the following form:

$$\sigma(\mathbf{z}) = \frac{e^{\mathbf{z_j}}}{\sum_{k=1}^{K} e^{\mathbf{z_k}}} \quad with\ j = 1, ..., K. \tag{2.10}$$

  Softmax is particularly useful in DL as it can be used in the output layer of a DNN to classify an input vector containing probability values and add up the elements into an output vector containing the final prediction.

2.1.4 Training Process and Optimization Techniques

In previous sections, we analysed how a DNN processes data throughout its frame. The whole learning process, however, considers several factors, such as functions and parameters, which have not been tackled yet. We can define

the *training process* as the iterative updating of weights and biases within the network in order to obtain as faithful a representation as possible of the input data set. This naturally implies fundamental importance in the quality of the dataset used for the training. A good dataset allows the network to capture the most important *features* for classification, allowing it to distinguish the various set's classes.

The network parameters are not updated randomly, otherwise achieving a generalization on the data would take an infinite amount of time considering that modern networks contain millions of parameters. This activity must therefore be done automatically and efficiently: we use a *loss function* L that computes the error of the model f with parameters ϑ, given the input/output pair $(\boldsymbol{x}, y)$:

$$loss = L(f(\boldsymbol{x}; \vartheta), y). \tag{2.11}$$

Our scope is to find the set of parameters ϑ which minimize the expected value of the loss function in the distribution of data μ, i.e. the *cost function*:

$$J(\vartheta) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}[L(f(\boldsymbol{x}; \vartheta), y)]. \tag{2.12}$$

At this point, we need to point out that our distribution set $\mathcal{D}$ is not the real distribution but just an approximation of it, named *empirical distribution.* Thus, we can define the *risk* as:

$$R(f) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - f(\boldsymbol{x}^{(i)}; \vartheta)), \tag{2.13}$$

with total number of samples n, $f(\boldsymbol{x}^{(i)}; \vartheta)$ and $y^{(i)}$ being respectively the i-th predicted label and the true label of the training set.

The risk-reducing operation is called *Empirical Risk Minimization (ERM)*:

$$\min_{\vartheta} R(f). \tag{2.14}$$

2.1.4.1 Backpropagation Algorithm

Considering a single training cycle, as we previously saw, we get an output based on internal computations of the DNN. Recalling the MLP algorithm, at

first the network's weights can be randomly instantiated or set to 0, thus they do not represent a generalization of the training set yet. To get the right values we need to compute ERM in a Deep Neural Networks context. Although, to perform a minimization operation we need to get the information about the steepness of the cost function at the current step with current parameters. This information is the *gradient* of the function itself, with respect to the set of parameters ϑ: we will better understand in Section 2.1.4.2 how we can use it properly.

By now, we just need to find a way to calculate it, since the final cost depends on every single vector weight in all the net. To solve the problem we can use *backpropagation*. This algorithm allows computing gradient of the cost function $J(\vartheta)$ for each neuron of the network with respect to its parameters.

As we have seen extensively in Section 2.1.2, the layers of the network are composed of several units, thus we rely on the *chain rule* to calculate the gradients. Given then a single weight w_{jk}^l, with l the current level in the network, the following formula shows a sigle step of the rule:

$$\frac{\partial J}{\partial w_{jk}^l} = \frac{\partial J}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial J}{\partial z_j^l}a_k^{l-1}, \tag{2.15}$$

and z_j^l can be defined as

$$z_j^l = \sum_{k=1}^{m}(w_{jk}^l a_k^{l-1} + b_j^l), \tag{2.16}$$

with m number of neurons in $l-1$ layer and a_k^{l-1} being the k-th neuron's activated output in layer $l-1$.

The same reasoning could be applied to the bias b_j^l, which being a constant does not contribute to the calculation:

$$\frac{\partial J}{\partial b_j^l} = \frac{\partial J}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial J}{\partial z_j^l}1. \tag{2.17}$$

From 2.15 and 2.16 one can easily get an idea of how many gradients need to be calculated during a single iteration of the algorithm. (**Note:** in practice with the mainly used libraries, the gradient of each vector is computed at the end of each forward step tracing an acyclic graph whose leaves are the input tensors and roots are the output tensors. In this way gradients are computed

automatically). Having introduced the preliminary aspects of DNN gradient calculation, we can now move on to the formulation of the algorithm on pseudocode. The element-wise multiplication is defined by $\odot$ operator, $\Omega(\vartheta)$ is a regularization parameter, and h represents the output of a designated hidden layer.

Algorithm 1 Backpropagation Algorithm

1: Compute the gradient of the loss with respect to the output layer:
$\mathbf{G} \leftarrow \nabla_{\hat{y}} L(\hat{y}, y)$
2: **for** $k = l, l-1, ..., 1$ **do**
3: Compute the gradient of the loss with respect to the layer's pre-nonlinear activation:
$\mathbf{G} \leftarrow \nabla_{a^{(k)}} J = \mathbf{G} \odot f'_A(a^{(k)})$
4: Compute the gradient of the loss with respect to the layer's weights and biases:
$\nabla_{b^{(k)}} J = \mathbf{G} + \lambda \nabla_{b^{(k)}} \Omega(\vartheta)$
$\nabla_{W^{(k)}} J = \mathbf{G}(h^{(k-1)})^\top + \lambda \nabla_{b^{(k)}} \Omega(\vartheta)$
5: Propagate the gradients of the loss with respect to the activations of the lower-level hidden layer's activations:
$\mathbf{G} \leftarrow \nabla_{h^{(k-1)}} J = (W^{(k)})^\top \mathbf{G}$
6: **end for**

In short, at first, the gradient of cost function J is calculated on itself. Then, following the network acyclic graph backward, J gradient is computed with respect to one of the parents according to 2.16. Doing the same kind of iteration, going upper in the network, we stop when we reach the target ancestor at level l. There may be a case in which an ancestor is reachable from the output node through different paths. In this case, it is sufficient to sum the gradients obtained from all the existent paths.

Now that it's clear how the gradient is computed in the DNN context, we can better understand how the optimization process works. In the following section, we furnish an in-depth explanation of how the cost function can be effectively minimized.

2.1.4.2 Gradient Descend

Optimization in ML and DL is the process in which we try to lower the cost function of a certain model by "inspecting" its shape. As introduced earlier, the cost function J is dependent on its parameters. In the case of neural networks,

it is therefore necessary to modify the internal weights of the model, on the basis of the gradient of the cost function $\nabla J(\vartheta)$.

One way to do so is through the *Gradient Descend algorithm (GD)*. As the name implies, at the end of each forward step of the network, the algorithm updates the parameters in the opposite direction of the gradient of cost function(previously calculated with backpropagation, as we have just seen in Section 2.1.4.1). We define this update as a *step*, taken in the *descending* direction of the cost function J, which has the following form:

$$\vartheta_s = \vartheta_{s-1} - \eta \nabla_\vartheta J(\vartheta), \tag{2.18}$$

where ϑ_{s-1} is the set of parameters at the previous step with respect to the current optimization process and η is a parameter called *learning rate*, and it's responsible for the size of the step we take in every iteration of GD. Learning rate value is a crucial component of optimization: if the value is too low the whole learning process will take too many forward steps to well generalize the training set, on the other hand, if it's too big, the risk is to move too fast in lowering the gradient resulting in a bad generalization.

Using the GD algorithm, the intention is to reach a minimum point, which we define as a *local minimum*. In general, it is not possible to know whether a correctly trained model reaches a *global minimum* since we do not know the whole cost function, but we can rely solely on its gradient to at least understand what it looks like in a neighborhood of the point constituted by the current set of parameters. When $J(\vartheta)$ converges, i.e. when the loss curve calculated in 2.11 for each forward step becomes flat and does not change, the training process is considered finished, and the model classification accuracy cannot be improved anymore.

In practice, pure GD as it is shown above can be computationally expensive, thus resulting in a time-consuming procedure, especially with modern large-scale learning problems which involve huge sample datasets. In this scenario other alternative ways to apply GD algorithm arises.

Instead of going through all examples at once, we may use *Stochastic gradient Descend (SGD)*. This GD-variation algorithm performs the parameter update after a specified number of samples, defining the *mini-batch* size m. Therefore, the proper learning phase happens every $|\mathbb{D}_{train}|/m$ steps, where $\mathbb{D}_{train}$ represents the cardinality of the training set. We can therefore define the

partial gradient of a mini-batch k, $\mathbf{G}^{(k)}$ as:

$$\mathbf{G}^{(k)} = \frac{1}{m}\nabla_{\vartheta}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\vartheta), y^{(i)}) \tag{2.19}$$

Thus, the update operation in SGD has the following form:

$$\vartheta_s = \vartheta_{s-1} - \eta\,\mathbf{G}^{(k)}. \tag{2.20}$$

It has been shown how approaching an area of local optima, SGD oscillates a lot while only making hesitant progress along the bottom [55]. This problem was overcome a decade later using *momentum* [43]. This technique speeds up effective regularization processes by exploring further the phenomenon of gradient "Falling down" to the nearest point of local minima. The method helps accelerate SGD in the relevant direction and dampens oscillations thanks to a "velocity vector" v, which is multiplied by each SGD step with a fraction of itself. The fraction is actually decided from momentum parameter γ The formula can be seen below.

$$\begin{cases} v_s = \gamma v_{s-1} + \eta\nabla_{\vartheta} J(\vartheta) \\ \vartheta_s = \vartheta_{s-1} - v_s \end{cases} \tag{2.21}$$

We can easily understand the momentum technique if we see the descending gradient as a ball: when it reaches steeper areas, it starts getting faster and faster in the direction of the local minimum with γ as acceleration parameter. As a result, we gain faster convergence and reduced oscillation.

Another way to get better optimization is using a learning rate *schedule*, an algorithm that tries to adjust the learning rate during training (using, for instance, annealing). This process is made according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to specific datasets. [44]

2.2 DNNs Analysis

2.2.1 Main Flaws

Over the last decade, deep learning has received a great deal of attention not only in research centers but especially in commercial contexts where data clas-

sification is increasingly common. In this reality, the need to achieve a high degree of precision in the performance of tasks by the algorithms has emerged. Over the years, this has led to an increasing focus on both the power to solve problems and the reliability of the results. Below are some of the most common issues encountered in the learning process of deep networks.

2.2.1.1 Overfitting and Underfitting

During the training process, the model tries to fit as better as it can training dataset, i.e. the algorithm aims to maximize the precision of output predictions. However, what is intended is something else, namely a good generalization of the input data. In other words, the model must be able to recognize the internal features of each class y without making direct comparisons with the images already analyzed. That being said, we might face two potential problems:

- **Overfitting**: The condition in which the prediction accuracy on the training set gets very high in the training phase, but then submitting the same model on new, fresh data (like on the test set) results in poor performance, making mainly wrong predictions. Usually, this phenomenon occurs because the model is too complex in relation to the training set, so a high number of parameters allows, by mistake, an exaggeratedly accurate classification. Another reason that can trigger overfitting is a too high number of training epochs[4] which causes a high variance in the model. This means that any variability in the data has been learned, even that which does not occur in the actual application.

- **Underfitting**: The opposite case to the previous one occurs when there are few parameters in the model and there is a high bias in the classification. This means that the learning process is too simple or has taken less time than necessary (i.e. the loss function has not yet converged). Another cause of underfitting can be an inaccurate training set, which does not highlight the features to be learned or has less data than the model needs.

In practical terms, it is possible to apply an intermediate phase between learning and testing, called the *validation phase*. This additional process uses a validation dataset to evaluate the performance of the model on different data from the training data while tuning the hyperparameters. Validation is a form

[4] An epoch, in the implementation phase, corresponds to a complete training cycle on the whole dataset

of *model selection*. the validation loss gives an indication of how well the model is generalizing the data, since it performs forward steps like training, albeit on different inputs. The result is a curve that can be compared with the shape of the training loss, there are two cases:

1. The validation loss has the same trend as the training loss, it starts high, goes down, and then converges in a hyperbolic way towards a similar circle to the training one. This information shows us that the model is generalizing well and is a good classifier even on new data.

2. The validation loss starts to converge and then rises exponentially, while the training loss converges. This behavior suggests that the model is overfitting, as it is not able to correctly classify data not present in the training set.

2.2.1.2 Vanishing and Exploding Gradient

When training on feedforward neural networks which utilize the gradient learning method with backpropagation seen in Section 2.1.4, we may witness a peculiar phenomenon strictly correlated to the optimization procedure. When calculating the gradients of all the matrices of the weights $\mathbf{W}$ in each level l of the network, these tend to become smaller and smaller, until at some point they no longer contribute in minimizing the cost function.

This event is called Vanishing Gradient, and is a common problem, especially when using very deep FNNs, as in the case of VGG [49]. The reason behind this behavior is in fact a direct cause of the number of iterations that the chain rule (also described in 2.15) performs during the execution of backpropagation.

To better understand, we give a mathematical representation. Consider a deep neural network with l layers and with parameters at each layer represented as $\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, ..., \mathbf{W}^{[l]}$. For simplicity let's state the following:

- A linear activation funtion, $f_A = z$
- Null bias, i.e $b^{[i]} = 0,\ \forall i : 0 < i \leqslant l$

Then the output of the model is of the form: $\hat{y} = \mathbf{W}^{[l]}\mathbf{W}^{[l-1]}....\mathbf{W}^{[2]}\mathbf{W}^{[1]}\boldsymbol{x}$. Let's define the weight matrix in level i as:

$$\mathbf{W}^{[i]} = \begin{bmatrix} n & 0 \\ 0 & n \end{bmatrix} \tag{2.22}$$

with $n \in \mathbb{R}$ and $n < 1$ except last layer. Than we then have:

$$\hat{y} = \mathbf{W}^{[i]} \begin{bmatrix} n & 0 \\ 0 & n \end{bmatrix}^{i-1} \boldsymbol{x}, \tag{2.23}$$

thus, it is easy to see that when l is big, and $\mathbf{W} < I$ then the gradient will tend to be 0. On the other hand, the opposite problem happens when $\mathbf{W} > I$, but in that case, we assist to the so-called *Exploding Gradient*, i.e. it becomes so big the model gets in the same bottleneck issues of the previous case.

2.2.2 How to solve instability: Regularization

In the previous Section we went through some problems that may occur during DNNs training. In the following, we present solutions to make Artificial Neural Networks more stable and less prone to common issues like the ones reported. The method of applying some additional operation to improve training is called *Regularization*.

2.2.2.1 Early Stopping

Let's suppose we start training a DNN while making validation after every fixed number of forward cycles. Keeping track of both training and validation loss, we realize at some point that the model starts to overfit. As we previously saw, or the model is too complex, or the data is not sufficient. One might wonder whether there is a way to obtain an acceptable model without necessarily having to change the reference dataset or make the model less complex. There are indeed some ways to make regularization, i.e. trying to keep the model stable while it is being trained and prevent overfitting. If the learner is using an iterative method (as in the case of the Gradient Descend) and the validation loss reaches an acceptable level (resulting in good classification accuracy), it is possible to use the *Early Stopping* technique to block the algorithm as soon as the curve, previously in convergence, starts to rise again. In this way, we can save the parameters that best fit both the training and validation data, thus the ones that make the best classifier with the fixed starting parameters, as soon as the performance starts to degrade. We could also see early stopping as a proper model selection action since we actively choose the model to keep based on evaluation parameters.

Note that all models if trained too long, sooner or later end up overfitting the training data, as the set is finite and the model has limited complexity. That being said, it is necessary to choose a trigger for the evaluation scheme. Choosing to stop as soon as the performance on the validation dataset decreases is certainly a good idea, but the fluctuations of the model must also be taken into account. It is, therefore, necessary to consider a suitably wide range to establish if the validation is getting worse [6].

2.2.2.2 Dropout

One of the main issues regarding DL when it was not a fully launched field yet, was mainly the propensity of DNNs to end up overfitting. Building up wider and deeper networks could just rise model complexity, what they were lacking in fact was a proper regularization technique. Some methods started to be used in the early 2000s, such as L1 and L2 weight penalties [54], however, they did not solve the problem completely. In fact, what happens in that condition is called *co-adaptation*: learning all the weights together makes some connections have more predictive capability than others. This means after a complete training phase some node connections are more trained and some are weaker, thus ignored by the algorithm most of the time, making the algorithm almost deterministic in weight choice.

A decade after, a revolutionary method called *Dropout* was introduced and it finally gave the tools to improve the field of Machine Learning. This technique manages to overcome the problem described above of networks focusing only on certain connections, without losing the main purpose, as a regulator, to prevent overfitting. The main idea of dropout is to use only certain connections in the network and to change, at each forward cycle, the nodes to be considered active and inactive. This is done by operating each time on a new network, derived from the architecture set up initially, removing some inputs or hidden nodes, along with their connections. To establish which elements of the original network must be discarded we use a dropout rate $\delta \sim \boldsymbol{B}(p)$, i.e. equal to the Bernoulli distribution. Therefore, δ is equal to 0 with probability p and 1 otherwise [5]. Since $p(1-p)$, the usual value to keep variable p is 0.5 for intermediate layers, which gives the maximum regularization possible. For input, layer should be set to 0.2 or lower, since dropping input data can lead to unexpected behavior. Setting $p > 0.5$ is instead not recommended, since it does not improve generalization and cut away too much of the network.

Dropout allows to produce a "minimized-loss" network the same way as a regularized network does, with all the benefits explained before.

Chapter 3

Training Paradigms

Machine Learning discipline comprehend a vast number of models able to generalize a task from a pre-made set of samples [2]. Each algorithm elaborates data using different statistical indices, and thanks to them, after a learning procedure, it can perform prediction on whatever new similar information to the input dataset. In this chapter, we discuss some approaches in defining the learning model by distinguishing not only the learning task we need to tackle but also what kind of data we possess. This analysis's fundamental since the learning process can involve different kinds of information based on what we actually know about the input set. The principle distinction on samples is based on classification knowledge. If input examples do not contain a classified label, it will be harder to generalize (i.e. find common patterns) in our dataset, mainly because we don't know how many different classes are actually contained in the input set, and also because the algorithm does not dispose of information that can help the optimization of the feature learning process. To summarize the different approaches, we can categorize three main learning techniques:

- **Supervised Learning:** it requires a dataset with labeled inputs,
- **Unsupervised Learning:** can perform learning tasks without the use of any label on samples,
- **Self-Supervised Learning:** a mixed paradigm between supervised and unsupervised which partially requires labels to execute.

The next sections of the chapter are dedicated to the deepening of the aforementioned techniques. Particular attention is given to self-supervision since it is the main topic of the study.

3.1 Supervised Learning

Supervised tasks in machine learning define the problem of generalizing and predicting patterns inside a set of homogeneous data. Applications of such a learning paradigm can be found in many data analysis contexts as computer vision, speech recognition, spam detection, information retrieval, and other specific disciplines. What distinguishes supervised learning from the other approaches is the use of labels to indicate the belonging category of each sample, in order to evaluate the prediction rate of the algorithm and perform the training step.

Given a distribution of input points $\mathcal{D}_{SL}$, we define the training examples as a set features $\boldsymbol{x}_i$ and labels y_i such that:

$$\mathcal{D}_{SL} = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_N, y_N)\} \ . \tag{3.1}$$

The goal of the learning process is finding a function $g : X \to Y$, with X denoting the input space and Y the output, i.e. label space. In practice there exist different g which satisfy the requirements, some more complex than others based on the number of parameters. The aim, in general, is to find a function of average complexity, to avoid cases of extreme complexity, that causes overfitting, and opposite too simple functions, turning into underfitting, i.e. poor generalization.

Many supervised models make use of a scoring function to determine an improvement of the learning process. This function can be defined as $f : X \times Y \to \mathbb{R}$ and it serves to better outline the target task:

$$g(\boldsymbol{x}) = \arg\min_{y} f(\boldsymbol{x}, y) \tag{3.2}$$

This generalization principle can be found in any supervised learning algorithm although it shows in different forms as probabilistic models. For instance, we recall the empirical risk minimization defined in Section 2.1.4 and Equation 2.13, which shows the principle of score function minimization applied in a deep learning context. Anyway, there exist also other kinds of self-supervised algorithms. For a deeper comprehension we report some of them:

- **Linear Regression:** learning model which assumes the existence of a linear relationship between the vector of input samples $\boldsymbol{x}$, also called *regressors*, and each dependent variable, i.e. each label y.

- **Logistic Regression:** it allows to solve classification problems using the *Logistic function* $f_{log} \in [0, 1]$, that is the same Sigmoid function defined in 2.9, to predict the outcome of input data. The criterion used to minimize the risk is the *maximum likelihood estimation (MLE)*, which we refer to [39] for an in-depth explanation of how it works.

- **Support-Vector Machines (SVM):** lies in the definition of *hyperplane* in order to set boundaries that help the classification of input data. Points that lie on the same hyperplane are identified as belonging to the same category. While logistic regression "compresses" points within the range $[0, 1]$, the SVM threshold is set as $[-1, 1]$ and serves to identify sample classes. The optimization process aims at maximizing the margin between data points and the hyperplanes. We report the *Hinge Loss function* without any other regularization, to give an expression of cost in the SVM context:

$$c(\boldsymbol{x}, y, f(\boldsymbol{x})) = \begin{cases} 0 & \text{if} \quad y \cdot f(\boldsymbol{x}) \geq 1 \\ 1 - y \cdot f(\boldsymbol{x}) & \text{otherwise} \end{cases} \tag{3.3}$$

Finally, the most effective and powerful expressions of the supervised learning paradigm are deep neural networks. To avoid useless repetitions, we recall Chapter 2, where we gave an in-depth and accurate explanation of the training process.

3.2 Unsupervised Learning

Supposing we have a large amount of chaotic data on which we want to make generalizations to classify fresh data inside the same macro-group. Previously supervised paradigms would not be able to perform such a task since this problem lacks the knowledge about the actual classes in which our data is semantically divided. Instead, what we can do is to find some patterns that may allow us to distinguish every input point in a finite number of classes, thus giving data some meaning by just analyzing its properties: in other words, *clustering*. This is a quite hard task to perform, in terms of complexity, time of execution, and quality/accuracy of the obtained results.

For clarification purposes only, we give a brief explanation on how some of these algorithms works, so that we can make an idea of the computation steps and overall complexity:

- **K-means Clustering:** it relies on setting an arbitrary number of clusters we expect to find using parameter K, which denote the number of *centroids*, i.e. the reference points in the output space which denotes each cluster region [17]. We define the centroid set as:

$$\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_K \tag{3.4}$$

  Moreover, some requirements need to be satisfied in order to consider the algorithm correct:

  – The union of clusters must contain all starting items,
  – Each item belongs to one and only one cluster,
  – Each cluster must contain at least one item and no cluster can contain all items.

  K-means aims at minimizing the following cost function:

$$f_{Kmeans}(\mathcal{C}) = \sum_{i=1}^{K} \sum_{X_j \in \mathcal{P}_i} ||X_j - \mathcal{C}_i||^2 \,, \tag{3.5}$$

  where P_i denotes the i-th partition over the set $\mathcal{P} = \mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_K$ and X_j is the j-th item of the input set.

- **Principal Component Analysis (PCA):** based on linear transformations of input variables, it defines a new space system in which the original component with most variance is settled on the first of the axes, the second biggest-in-variance component is settled on new second axes, and so on. We recall the study [59] to deepen the aspects of this unsupervised method since it is a complex algorithm that is not necessary for experiments implementation.

3.3 Self-Supervised Learning

Previously learning methods are an attempt to achieve the same goal in two opposite ways. Having label data can surely make the workflow easier and returns back effective and accurate models, but is still too much prone to adversarial examples due to its feature learning structure. Furthermore, labeling data as already pointed out in Section 1.1.2, can be a long process, especially

on modern dataset sizes. On the other side, clustering thousands of inputs with unsupervised techniques leads to less precise inputs grouping, which needs the intervention of a human to interpret and perform the actual labeling. If we also take into account the computationally expensive process, it gets even less useful in terms of results than the supervised approach.

As the need for more advanced techniques raised, in past years came up the concept of *Self-Supervised Learning (SSL)*: this method allows learning effective feature representations without the need for labels. To be more precise labels are just used in certain parts of the framework, to initialize network weights and calculate accuracies [11]. Despite this property which recalls the unsupervised method, self-supervised takes advantage of the same supervised deep neural network architecture. What really makes a difference is the "wrapper" logic part that encloses such networks, which usually includes some data processing such as *data augmentation*[1] or performs some particular elaboration of the results thus obtained.

State of the art in deep learning brings us different paradigms when it comes to self-supervised models. As just stated these frameworks share the majority of their entire architecture: the goal is to perform training on a deep neural network, and as a result have a generalized model in the form of a deep neural network with preset weights. Thus it is possible to perform different feature learning processes changing the outlined framework. The most common and effective paradigms, also described in [33] are the following:

- Generative Learning
- Contrastive Learning [8] [15]
- Discriminative Learning

Given a brief introduction of self-supervised methods, we are now going to dive deep into the concept of Contrastive Learning, since is the fundamental tool with which the experiments were conducted.

3.3.1 Contrastive Learning

Recent self-supervised models managed to reach performances that have overcome some of the best supervised algorithms used so far. Most of this success can be attributed to *Contrastive Learning*, which is a paradigm to perform

[1] the process in which a slight modification is applied to input data to increase the number of samples in a dataset, or to prepare the model to perform some specific feature selection.

high-feature learning using a paired comparison between examples. It is inspired by the way humans recognize unknown objects by correlating their distinctive features with similar ones also present in objects they already know. This characteristic is defined as prior knowledge.

The main idea around contrastive learning is defined by the splitting task between data which is similar and data that is dissimilar, for any given sample inside a distribution $\mathcal{D}_{SSL}$, i.e. the reference dataset. This criterion overcomes the strict label-referring task of pure supervision, to deepen more what features matter for the final classification activity.

Let's consider an input sample $\boldsymbol{x} \in \mathcal{D}_{SSL}$, we can thus define the contrastive task as the attempt to learn an encoder $f_{enc}(\cdot)$ through the comparison of other similar inputs, called positive samples $\boldsymbol{x}^+$, and dissimilar ones, i.e. negative samples $\boldsymbol{x}^-$, such that:

$$sim(f_{enc}(\boldsymbol{x}), f_{enc}(\boldsymbol{x}^+)) \gg sim(f_{enc}(\boldsymbol{x}), f_{enc}(\boldsymbol{x}^-)) \tag{3.6}$$

where *sim* is a similarity metric, essential to compute how much a sample pair is semantically consistent with itself. It can be, for instance, the dot product or any other kind of algebraic distance between vectors [3].

Then, in order to properly train our classifier, we formally define the loss function in order to push the learning process in labeling positive pairs within the same class (and thus preventing miscalssifications with negative pairs):

$$\ell_N = -\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}_{SSL}}\left[\log \frac{\exp\left(f_{enc}(\boldsymbol{x})^\top f_{enc}(\boldsymbol{x}^+)\right)}{\exp(f_{enc}(\boldsymbol{x})^\top f_{enc}(\boldsymbol{x}^+)) + \sum_{k=1}^{N-1}\exp(f_{enc}(\boldsymbol{x})^\top f_{enc}(\boldsymbol{x}_k))}\right], \tag{3.7}$$

where numerator takes into account positives with respect to input $\boldsymbol{x}$, whereas denominator contains the totality of N examples, divided into the positives and the remaining $N-1$ negatives. The factors in 3.7 are to be considered as the product of the encoder and a pooling operation[2]. ℓ_N is a particular loss used specifically for contrastive learning, which is commonly named *InfoNCE loss*.

Now the just explained procedure includes the basic elements composing the learning process through contrastive samples. Nonetheless there exist many different kinds of approaches when dealing with this paradigm, as many past studies highlighted [4], [20], [22], [42]. However, one contrastive framework

[2]operation which aims to simplify the semantic meaning of resulted data, from computations such as a base encoder $f_{enc}(\cdot)$

stood out by considerably improving performances of self-supervised technique. In the next Section we introduce the model we reproduced to conduct the experiments on the thesis, the so-called *SimCLR*.

3.3.2 SimCLR

3.3.2.1 Introductory Notes

Many state-of-the-art pieces of research have proven the efficacy of self-supervised learning, especially in terms of feature representation quality. SimCLR is one of the best explanations of this approach: it provides a simple and intuitive framework that applies all the notions about contrastive learning pointed out in Section 3.3.1, with some more intuitions that improve the model performance. SimCLR was presented in 2020 with the paper *"A Simple Framework for Contrastive Learning of Visual Representations"* from Google Research [7].

To better examine how the whole process works, let's separate the different logic parts in which the algorithm is composed. We can distinguish 3 main sequences:

1. Mini-Batch creation and Augmentation of Samples
2. Deep Neural Network Processing and Feature Representation
3. Feature Learning with Loss Minimization

Step 1 is executed only at the beginning of the algorithm, and it is necessary to properly organize samples for feature learning. Then, steps 2 and 3 iterate through each mini-batch of the dataset and as many times as the number of predetermined epochs. After that, if the other learning parameters were well set and evaluation metrics on the trained model are good enough, it is ready to be used on similar tasks.

3.3.2.2 Mini-Batch creation and Augmentation of Samples

First to manage modern dataset sizes, a proper splitting of all the input samples is required. Similarly to what happens for common supervised techniques, we separate data into different mini-batches in a random manner:

$$\mathcal{D}_{SSL} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_M\} \,, \tag{3.8}$$

such that each batch has same size N, resulting in $\frac{|\mathcal{D}_{SSL}|}{N} = M$ different batches.

One of the core aspects introduced in Section 3.3.1, is the concept of positive and negative samples. SimCLR incorporates this principle by using data augmentation on each mini-batch $\boldsymbol{x}_k$: every sample is augmented with different techniques such as random cropping, resizing, flipping, color distortion, and also by applying image noises. This action is made twice in order to create two different augmentations from the original input, as shown in Figure 3.1. These new samples are then positioned within the original order consecutively in a new mini-batch containing $2N$ data points. Given an input mini-batch $\boldsymbol{x}_k$, data augmentation will generate two views from the original data, which we recall as $\tilde{\boldsymbol{x}}_i$ and $\tilde{\boldsymbol{x}}_j$.

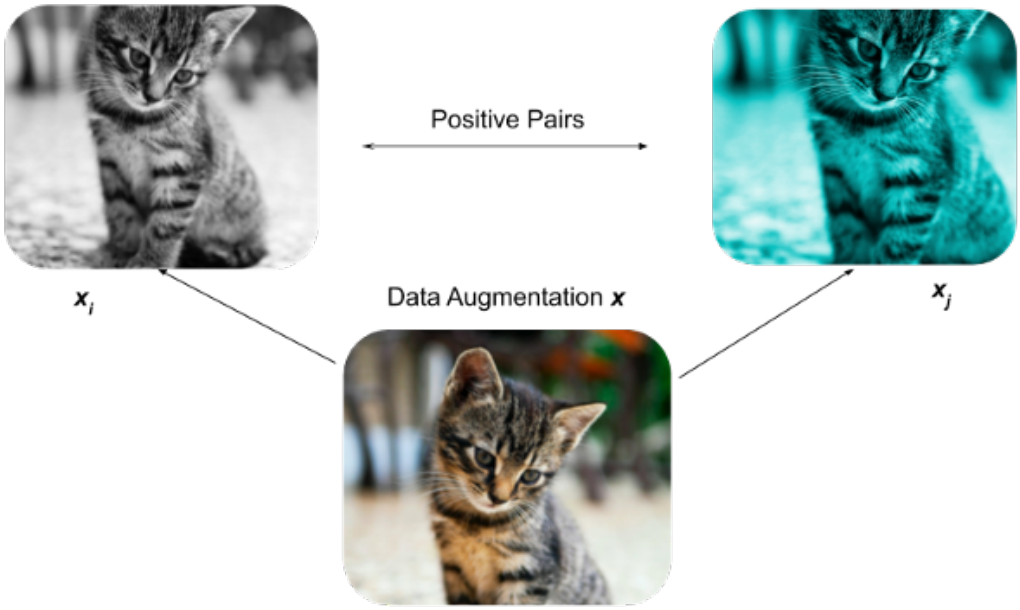


Figure 3.1: Positive Pair Creation [57]

SimCLR does not define positive and negative examples explicitly, instead, it takes advantage of the new mini-batch created to restrict the concept on the encoding part of the algorithm. In a nutshell, the model will *de facto* consider each augmented pair as positive and all the other $2(N-1)$ samples as negatives. However this distinction does not take place in this sequence of the algorithm, but it is part of the optimization process as we will cover next.

3.3.2.3 Deep Neural Network Processing and Feature Representation

Now that is clear how the model prepare the sample data to be processed, we introduce the encoder $f_{enc}(\cdot)$, i.e. the neural network, to define the actual learning strategy. In practical terms, SimCLR allows the implementation of different kinds of architectures. Nevertheless, in this work, we follow the author's choice of using *ResNet* architecture, a deep residual neural network with good performances [18], [19]. Let's consider again an input mini-batch $\boldsymbol{x}_k$, given its augmented views to the encoder we receive the following output after the average pooling process:

$$\begin{cases} \boldsymbol{h}_i = f_{enc}(\tilde{\boldsymbol{x}}_i) \\ \boldsymbol{h}_j = f_{enc}(\tilde{\boldsymbol{x}}_j) \quad \text{with} \quad \boldsymbol{h}_i, \boldsymbol{h}_j \in \mathbb{R} \, . \end{cases} \tag{3.9}$$

The results thus obtained do not represent the final features to be optimized. We need one more computation to apply the contrastive loss, which is basically what makes the model learn the input features based on positive and negative samples. In order to map the points $\boldsymbol{h}_i, \boldsymbol{h}_j$ we introduce the *projection head* $g(\cdot)$, which is simply an MLP with a single hidden layer and a ReLU function (we recall Section 2.1.2.2 and Section 2.1.3.1 where we deeply covered the topics):

$$\begin{cases} \boldsymbol{z}_i = g(\boldsymbol{h}_i) = \boldsymbol{W}^{(2)} ReLU(\boldsymbol{W}^{(1)} \boldsymbol{h}_i) \\ \boldsymbol{z}_j = g(\boldsymbol{h}_j) = \boldsymbol{W}^{(2)} ReLU(\boldsymbol{W}^{(1)} \boldsymbol{h}_j) \end{cases} \tag{3.10}$$

The reason behind the use of a projection head lies in the representation quality of the given output. In fact, by leveraging $g(\cdot)$ it is possible to keep a larger amount of information, by just compressing the images into a latent space representation to give a better weight on crucial features. This fact makes better performance on the contrastive loss operation, which has been empirically proven by the authors.

3.3.2.4 Feature Learning with Contrastive Loss Minimization

As previously stated, the purpose of the contrastive loss function is to affect the learning process to make positive pairs (i.e. same feature data) to attract and negative ones to reject each other. In order to perform this step, it is necessary for this task to define the concept of similarity between samples. Calling

back the fact we treat input data as vectors, it comes convenient to use *cosine similarity* of output representation to express how much different two samples actually are in terms of features. It is indeed trivial to realize that similar pieces of information in the representation space are closer in an algebraic perspective than different ones. According to 3.7 we adapt the contrastive loss function for a subset of inputs $\{\tilde{\boldsymbol{x}}_k\}$ to define the contrastive prediction task, using *Normalized Temperature-scaled Cross-Entropy Loss (NT-Xent)*:

$$\ell_{i,j} = -\log \frac{\exp(sim(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(sim(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)} , \tag{3.11}$$

where $sim(\boldsymbol{z}_i, \boldsymbol{z}_j)$ describes the previously cited cosine similarity

$$sim(\boldsymbol{z}_i, \boldsymbol{z}_j) = \frac{\boldsymbol{z}_i^\top \boldsymbol{z}_j}{||\boldsymbol{z}_i|| \cdot ||\boldsymbol{z}_j||} , \tag{3.12}$$

and $\mathbb{1}_{[k \neq i]} \in \{0, 1\}$ has value 1 if and only if $k \neq i$, and serves to identify only negative pairs respect to positive. Moreover, constant τ represents a *temperature* parameter.

The optimization process is crucial in performing an efficient visual representation, in fact, it represents the evaluation factor in feature processing. It is possible empirically to track the distance of positive pair representations during training: what can be seen is an initial wide distance between positive pairs, which gets smaller and smaller the more the algorithm keeps optimizing its feature visualization on dataset classes. This trivial mathematical behavior translates into an architecture that generalizes a set of classes without the need of using any label, as we just described in these last sections.

3.3.2.5 Last Remarks

SimCLR is without doubt an effective self-supervised learning model, which exploits different math and image processing concepts to reach good generalization rates for specified tasks. There are some remarkable notes to point out to better understand how architecture works. It was proven as to how much important is the role of data augmentation in pair generation at the beginning of the algorithm: in particular, a combination of augmentations turns out to be indispensable to make SimCLR work properly at its best, in fact, single transformations did not get good enough representations during testing of the

model. For instance, cropping and color distortion is crucial and necessarily need to be included in the augmentation process.

Furthermore, mini-batch size N turned out to be another fundamental property in defining SimCLR effectiveness. Performing on the same training tasks but with different sizes (from 256 to 8192) the authors found out that generalization benefits more by bigger batch sizes than smaller ones. This result can be found also in other kinds of models [51], and it is correlated with the fact that batch management is compulsory since state-of-the-art architectures can not receive directly very large datasets entirely at once, for a matter of space and computation complexity. Another consideration can be made regarding mini-batch sizes: in contrastive logic of positive and negative pairs, there exists one flaw which lies in having casually a lot of samples of the same class inside one batch. In this scenario what happens is a lot of samples classified as negatives by NT-Xent loss should have been classified as positive instead. In the original version of the paper, this problem was not tackled in any way. Thus, using large batch sizes could also benefit in terms of generalization due to the fact it was less probable such an event could happen.

Having covered the main different aspects of deep learning and defined the respective tasks, we can now move on to the next fundamental topic of this thesis. In the next chapter, we are going to understand accurately how adversarial learning works, and its effect on the training process of the network.

Chapter 4

Adversarial Training

In Section 1.1.1 we gave an overview about what adversarial examples are and the main problems correlated to these data, which basically shows the limitation of trained models with current deep learning techniques. Past research has shown how adversarial training can be a solution to effectively overcome this flaw. The process does not represent a cost-free solution though: as we deeply acknowledge on experimentations, robustness comes with a trade-off in accuracy on general predictions. We link to the Chapter 5 for more detail about this behavior.

In the next section, we provide some general ideas to introduce adversarial techniques.

4.1 Basic Principles

Since non-robust models are subject to classification errors on adversarial samples, what we need is to tackle the problem before it shows up on a completely trained architecture. The core idea when it comes to adversarial training lies in feeding the network with adversarial samples so that it can better generalize and produce robust features. In a common situation, we do not have these kinds of samples, or we do not dispose of a large amount of them, thus the only way to perform robust training is to artificially generate adversarial examples.

The process of producing adversarial samples is called *attack*, meaning we are preparing every single sample, in order to challenge the network more in finding out solid representations for our dataset. The attack phase generally happens right before the samples are given to the model, and this is important to notice since in most cases we need some additional information to perform the actual attack.

Before getting into the proper adversarial generation mechanism, it is crucial

to show all the capabilities we can exploit using this method. Firstly, when it comes to deciding how to attack a model, it is definitely useful to make *threat modeling*, which means to analyze how it performs training to better choose how to design the attack [38]. We can subdivide attacks into two main principles:

- **Black Box:** An attack that does not consider any information about how the model works. This kind of attack provides a general way to return adversarial examples without expensive computations, nonetheless, it is obviously a weak adversarial process, since does not target specific features.

- **White Box:** When attacking procedure takes into account model information (usually parameters at each step, like gradients). In this way, it is possible to apply a more effective attack, "confusing" the model right in the feature learning process. White box techniques are way more effective than a black box, but they come with a higher computational cost. Usually, since training time matter less than the robustness of a model, these kinds of attacks are preferred with respect to black-box ones.

What really matters when we are facing the problem of designing adversarial training is the meaning of the applied attack. As we largely deepen in Chapter 2, training is the process of generalizing data starting from inherent elements which give each sample a defined place into the target space. This elaboration can be defined also as the model capability to acquire the *semantic* of each sample. This property is essential to analyze what makes some models more robust than the others, that is the flexibility in identifying what features are critical to classify correctly a given label, and what is not. Designing an attack that makes the model fail on important sample features instead of on random parts of it, can twist the performances completely.

Let's consider for instance an attack that applies random noise to an image, or like it is described in [23], an attack that turns each image negative. It is easy to realize how applying those kinds of changes does not necessarily make the model more robust: in fact, in the first case, we don't know how much noise we need to apply to make the model fail, and disrupting the signal too much can translate in losing the fundamental features of the sample, which we really don't want to happen to let the parameters set up in a useful way. In the second case, most of the information (like shapes and contrast) is kept, but every feature based on colors is completely lost. This can be useful to make the model understand better, for instance, shape features instead of color

ones. However, in some datasets may exist examples in which color does not necessarily matter (it is the case for digits or cars, to mention a few), and in those cases, the attack turns out to be useless. Both the attacks try to target in a general way the images, this property makes them black-box attacks, which can in some cases affect the semantics of the samples, but not necessarily, and this translates in a poor robustness improvement.

These reasons are just a few to explain how black box attacks are not very effective in order to get back a more reliable model. It is therefore a solid solution to check how the model behaves during training and apply some challenging variations to the architecture input. In the next Section we present the core idea to make white box adversarial attacks.

4.2 White Box Attacks

Adversarial training concept can be applied to every ML model which considers a "step by step" procedure in order to rich a proper prediction model. From now on we focus on deep learning environment on images, being the main subject of our analysis. Recalling the training process described in Section 2.1.4 let's consider a data distribution $\mathcal{D}$ over pairs of data samples $\boldsymbol{x} \in \mathbb{R}$ and label y. The main goal of training is to perform Empirical Risk Minimization on the cost function, as described in the formula 2.13. However, this process does not consider the possibility of encountering adversarial examples. What can help in finding a solution is inspecting how the model works to improve its cost function to generalize data: when dealing with deep neural networks the ERM process can be synthesized by backpropagation and gradient descend algorithms (we recall Sections 2.1.4.1 and 2.1.4.2 as a reminder of how they work). In fact, gradients are the information we are looking for when it comes to challenging the network since it gives the notion of the steepness and magnitude of the cost function at each iteration step. Normally to minimize the risk we would go in descending direction, thus updating the weights based on2.18, whereas in this context we are just interested in the gradient value associated with the targeted input $\boldsymbol{x}$.

4.2.1 Fast Gradient Sign Method (FGSM)

What we can do in practice to perform adversarial training, is generate a perturbation $\delta \in \mathcal{P}$, being $\mathcal{P}$ a set of allowed perturbations, to apply on an image $\boldsymbol{x}$ based on the magnitude of the gradient, so that the next iteration of the

algorithm will generalize less our input. The adversarial (i.e perturbed) image is on the form $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \delta$. This action can be interpreted as a "loss maximizer", translated in the image with a slight modification, which has to stay smaller than the precision of the features to not lose too much of the information. As suggested [34], it is a way to augment the ERM process, in order to improve the model in terms of robustness.

One simple application of the concept just explained is represented by *Fast Gradient Sign Method (FGSM)* [16] which basically computes the perturbation accordingly to what prevously mentioned:

$$\delta = \epsilon sign(\nabla_{\boldsymbol{x}} J(\vartheta, \boldsymbol{x}, y)) \ , \tag{4.1}$$

with ϵ being a small enough coefficient allowing the classifier to assign the same class to both $\boldsymbol{x}$ and $\tilde{\boldsymbol{x}}$. Now consider the dot product between weight vector $\mathbf{W}$ and an adversarsarial sample $\tilde{x}$:

$$\mathbf{W}^\top \tilde{\boldsymbol{x}} = \mathbf{W}^\top \boldsymbol{x} + \mathbf{W}^\top \delta \ , \tag{4.2}$$

thus the adversarial perturbation causes the activation to grow by a factor $\mathbf{W}^\top \delta$. Using *sign* function in 4.1 we are basically maximizing, in the same direction of the gradient, the amount of δ which is then handled by ϵ in order to adapt the attack based on our specific classification problem. The algorithm is better explained by the pseudocode below:

Algorithm 2 Fast Gradient Sign Method (FGSM)

1: Compute the loss from model logits with respect to the true class:
$L(\hat{y}, y) = f_{crit}(\hat{y}, y)$
2: Calculate Backpropagation algorithm (2.1.4.1) respect to the loss, in order to get input gradient:
$\mathbf{G}_{\boldsymbol{x}} \leftarrow \nabla_{\boldsymbol{x}} J(\vartheta, \boldsymbol{x}, y)$
3: Compute the perturbation with respect to input gradient sign, as described in equation 4.1. A factor ϵ allows to choose the amount:
$\delta = \epsilon sign(\mathbf{G}_{\boldsymbol{x}})$
4: Generate the adversarial example by adding the perturbation to the input:
$\tilde{\boldsymbol{x}} = \boldsymbol{x} + \delta$

The product of such an algorithm is an image with a small amount of noise, but way more effective than the black box noise described in Section 4.1. In

fact, the noise produced by FGSM manages to fool the classifier with a smaller amount of it, but still in a generalization-oriented environment.

FGSM works well in practice in many conditions, but it presents some flaws, especially concerning the architecture on which adversarial training is performed. For instance, the capacity of the network can help increase training performances, furthermore, large ϵ values do not guarantee a robustness increase. For these and other reasons in past years, some studies found alternative ways to implement this adversarial algorithm, with variations like the iterative version (I-FGSM) or the targeted version (T-FGSM). We report one of them for further investigations on the topic [30].

4.2.2 Projected Gradient Descend (PGD)

Having understood the main blocks composing the adversarial examples generation, it is now possible to analyze a different solution than the simple FGSM approach. What emerges from the evaluation of the method above is it takes into account just a single step of the function for each adversarial input created. This means the attack we are going to apply to the image in this way is still poor in terms of quality compared to other methods. Also, the previous approach does not take into account the trend of model loss, but only cares about the best perturbation to apply to fool the model. What makes a difference instead when designing robust classifiers, is considering the entire architecture and the training factors, and trying to put the less modification possible still improving performances.

Since we aim to reach the condition just described, we are going to optimize the process the more we can. Let's define the problem as the need to minimize the model loss with a set of parameters ϑ, and at the same time "soiling" the input using some δ from a set of perturbations $\mathcal{P}$. This formulation describes an adversarial learning problem, the so-called *saddle point problem* which is defined by:

$$\min_{\vartheta} \rho(\vartheta), \quad \text{where} \quad \rho(\vartheta) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}} \left[\max_{\delta\in\mathcal{P}} L(\boldsymbol{x}+\delta, y; \vartheta) \right] . \tag{4.3}$$

This is an example of both non-convex and non-concave problems. Convergence is not a trivial condition for such a matter, however, it is in practice possible to find acceptable solutions to the saddle point problem with some conditions. For example, the large capacity architecture cited before, using some effective first-order information [50], i.e. the gradient in our case of study, will be enough to

find the right perturbation, with a cost in terms of time compared to previous methods, as the problem tackled here belongs to a different scale of complexity.

One algorithm that turns out to be very effective in solving the aforementioned problem is the *Projected Gradient Descend (PGD)* algorithm, which as the name suggests, takes advantage of the GD procedure to solve 4.3. To better understand how the algorithm works, the pseudocode of PGD is provided ahead.

Algorithm 3 Projected Gradient Descend (PGD)

1: Choose a constrain set $\mathcal{Q}$ (as example here, ℓ_2-norm) and pick an initial random perturbation which satisfies it:
$\mathcal{Q} = \{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x}_0\|_2 \leq \epsilon\}$
$\boldsymbol{x}_0 \leftarrow \delta \quad$ with $\quad \boldsymbol{x}_0 \in \mathcal{Q}$
2: **for** $k = 1, 2, ..., K$ **do**
3: Calculate the estimated gradient and pick the descend direction for the input: $-\nabla f(\boldsymbol{x}_k)$
4: Update the input value based on step size σ:
$\boldsymbol{x}^*_{k+1} \leftarrow \boldsymbol{x}_k - \sigma \nabla f(\boldsymbol{x}_k)$
5: Use the projection as optimization step :
$\boldsymbol{x}_{k+1} = \arg\max_{\boldsymbol{x} \in \mathcal{Q}} \|\boldsymbol{x} - \boldsymbol{x}^*_{k+1}\|_2 \leq \epsilon$
6: **end for**

As can be seen in the algorithm structure, PGD requires some parameters to function properly. Each of them affects the quantity and the quality of the final perturbation, thus the adversary output generated:

- **Constraint Set:** is a set of points within which the algorithm is allowed to move the initial input. This means it defines how much perturbation can be applied to the image at each PGD iteration.

  There exist different types of effective sets, which usually define a "ball" around the target sample. The most common constraints are the following:

  - ℓ_2 is also called the Euclidean norm. It is defined as the distance of the vector coordinate from the origin of the vector space. For, instance, the ℓ_2-norm of a vector $\boldsymbol{x} = (x_1, x_2, x_3)$ is calculated as

$$\|\boldsymbol{x}\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2}\,, \tag{4.4}$$

in other words is the square root of the sum of squared vector values.

– ℓ_∞ express the largest magnitude among each element of a vector, which can simply be defined for a vector $\boldsymbol{x} = (x_1, x_2, x_3)$ as:

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i| \tag{4.5}$$

- **"Epsilon" parameter** (ϵ): which is the most important parameter in deciding how much perturbation we want to apply to the input. It basically sets the radius of the ball centered in the current sample $\boldsymbol{x}$, within which we can pick a perturbation to perform the PGD step.
- **Step size:** it is the magnitude of the attack step taken in direction of the gradient.
- **Iterations** (K): the number of PGD attack steps to be performed to return the desired amount of perturbation.

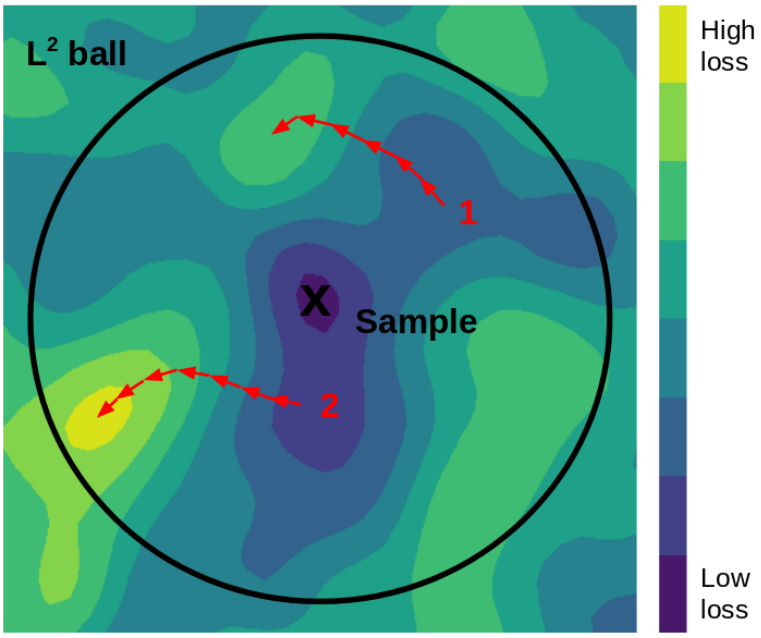


Figure 4.1: Projected Gradient Descend Visualization: the input sample is located in a low loss area. Segments 1 and 2 define the paths taken by two different perturbations inside ℓ_2-norm constraint. [28]

To summarize, what the algorithm does, is just choose a point around the ball defined by constraint set $\mathcal{Q}$, then iteratively (based on the chosen number of iterations K) step in direction of the greatest loss. Then, if the perturbation falls

out of $\mathcal{Q}$, PGD projects it back to satisfy the constraint. A visual representation of the process is shown in Figure 4.1.

Chapter 5

Experiments and Data Analysis

In Chapter 1 we already remarked the aim of the thesis as a careful analysis on deep neural networks, in particular in terms of robustness. Having understood the main functioning of the training process and having acquired the advantages related to self-supervision technique and adversarial training, we can now delve into the topic with experiments that will help to understand the dynamics that regulate models' robustness. If interested in looking at the source code, we created a GitHub repository of the project available [48].

5.1 Datasets

When preparing the experiments, in data analysis the choice of the dataset to use is crucial especially in designing the problem, since it is a key aspect when it comes to evaluating the model. As explained in Section 3.3.2, our model makes use of SimCLR contrastive architecture. Since we implemented the same encoder as the authors, we chose to make use of two datasets also employed by T.Chen in [7], *CIFAR10* and *CIFAR100*[1]. These datasets contain, as the names suggest, respectively 10 and 100 classes, each composing a set of 60000 color images 32x32 in size. The total amount is then divided into 5 train batches and one test batch:

$$\mathbb{D}_{CIFAR} = \{\mathbb{D}_1^{Train}, \mathbb{D}_2^{Train}, \mathbb{D}_3^{Train}, \mathbb{D}_4^{Train}, \mathbb{D}_5^{Train}, \mathbb{D}^{Test}\} \tag{5.1}$$

such that each batch contains 10000 samples. It is important to notice that training batches are not distributed homogeneously and may differ in the number of samples per class.

[1] `https://www.cs.toronto.edu/~kriz/cifar.html`

That being said we have CIFAR10 with 6000 images per class, which we report in code form for simplicity:

```
CIFAR10_classes = ['airplane', 'automobile', 'bird', 'cat',
   'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

Then, CIFAR100 contains 600 samples for each of the 100 classes, gathered into 20 sub-groups based on common features:

```
CIFAR100_classes = [
       'apple', 'aquarium_fish', 'baby', 'bear', 'beaver',
          'bed', 'bee', 'beetle',
       'bicycle', 'bottle', 'bowl', 'boy', 'bridge', 'bus',
          'butterfly', 'camel',
       'can', 'castle', 'caterpillar', 'cattle', 'chair',
          'chimpanzee', 'clock',
       'cloud', 'cockroach', 'couch', 'crab', 'crocodile',
          'cup', 'dinosaur',
       'dolphin', 'elephant', 'flatfish', 'forest', 'fox',
          'girl', 'hamster',
       'house', 'kangaroo', 'keyboard', 'lamp', 'lawn_mower',
          'leopard', 'lion',
       'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle',
          'mountain', 'mouse',
       'mushroom', 'oak_tree', 'orange', 'orchid', 'otter',
          'palm_tree', 'pear',
       'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy',
          'porcupine',
       'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket',
          'rose',
       'sea', 'seal', 'shark', 'shrew', 'skunk', 'skyscraper',
          'snail', 'snake',
       'spider', 'squirrel', 'streetcar', 'sunflower',
          'sweet_pepper', 'table',
       'tank', 'telephone', 'television', 'tiger', 'tractor',
          'train', 'trout',
       'tulip', 'turtle', 'wardrobe', 'whale', 'willow_tree',
          'wolf', 'woman',
       'worm'
   ]
```

For our research, it is important to notice that some classes contained in CIFAR100 may have similarities with others in CIFAR10 in terms of feature representation. For instance, it is reasonable to think the class 'wolf' shares some features with 'dog', like 'tiger' with 'cat', or 'bus' with 'truck' and 'car'.

It is indeed trivial identifying a classification problem on CIFAR100 as more complex than a corresponding one on CIFAR10, which we would expect to find also in experimental results in the form of an accuracy drop.

5.2 Project Set up

In order to conduct efficient classifications in terms of computational complexity and timing, we tried different training settings and found a reasonable compromise in using a mini-batch size of 256 for both CIFAR10 and CIFAR100, as they share the same number of samples per set type. Except for mini-batch partitioning, which is mandatory for running our model, we don't apply any further modification directly on the input sets, as being both balanced allows us to get consistent results. The only exception is made for contrastive data augmentation, which is secondly made inside the training process as explained in Section 3.3.2. According to [7] we use the following transformations:

- Random cropping,
- Random horizontal flipping,
- Color jittering with probability $p = 0.8$,
- Random grayscale with probabiliy $p = 0.2$,
- Gaussian Blur with a kernel size of a factor 0.1 respect to sample size (i.e, 32 in our case).

Previous transformations were applied twice to generate the two views $\tilde{\boldsymbol{x}}_i$ and $\tilde{\boldsymbol{x}}_j$ described in Section 3.3.2.2, which turned the actual training batch size to 512.

Conducting a general analysis brought us in defining the workflow in the "wider" way possible: we first implemented the SimCLR contrastive framework and, on top of it, we built the adversarial samples generation mechanism. The hook landed in overriding the `forward()` method of ResNet architecture, to first generate the adversarial version of each input mini-batch, by running the PGD algorithm concerning the mini-batch itself. In this way, it was possible to perform an adversarial, self-supervised classification problem. Each run was made different from the others by the use of the PGD ϵ parameter, whose purpose was deeply described in Section 4.2.2. We found the tasks defined by CIFAR10 and CIFAR100 to behave similarly, in terms of learning convergence,

when using the same batch size, which led us to use the same learning rate during the tests, with adequate results.

To enhance model performance and further test our framework on different kinds of tasks, after training with the contrastive learning paradigm we fine-tuned the network on a standard classification problem. It is indeed useful since changing problems on a pre-trained model allows us to evaluate deeply the versatility and robustness of previous techniques. It was possible to apply fine-tuning by loading the contrastive model and freezing all the pre-trained layers. On top of them, we placed a linear layer that mapped the output space of SimCLR to the label space, to get the proper classification result. This process is also explanatory when inspecting feature representation behavior based on the problem to be tackled.

With some exceptions which we will see later, we standardized training parameters in order to correctly evaluate the model even considering the different datasets utilized.

1. **SimCLR Contrastive Learning**

   - Number of epochs: 500
   - Batch-size $\mathbb{D}_i^{Train} = 512$ (256 in two augmented views)
   - Learning rate $\eta = 0.0002$
   - SGD (Section 2.1.4.2) optimization with Adam (same learning rate η)
   - PGD parameters:
     - Step-size $\sigma = 1$
     - Iterations $K = 3$

2. **Standard Classification (Fine-Tuning)**

   - Number of epochs: 100
   - Batch-size $\mathbb{D}_i^{Train} = 256$
   - Learning rate $\eta = 8 \cdot 10^{-5}$
   - We further optimize the final linear layer with Adam (same learning rate η) while always keeping the rest of the network as it was loaded
   - PGD parameters unchanged, when robust fine-tuning is performed.

Having the potential of performing adversarial training even in fine-tuning is an important advantage: it is possible to check how adversary examples affect different tasks and what is best to use to get the most consistent model.

To further deepen our analysis, we built up a Feature Extractor in order to inspect layer weights of our trained models. We treat the deep network as a tree-like structure, where each node represents a specific step of the learning process. It is thus possible to capture the information stored in any of the nodes for comparing purposes. The potential of this technique in networks analysis context will be clear in Section 5.4.3.

In the next section, we are going to explain the experimental criteria to properly evaluate model robustness.

5.3 Evaluation Metrics

During all the training and test workflow some evaluation cautions were taken to avoid generalization errors, especially in semantics, such as overfitting. After training each model and fine-tuning based con ϵ parameter as explained in the previous section, we proceeded to record performances with test metrics. Every step was evaluated with different feedbacks, which we report for clarity:

- **Loss**: For each training cycle we plot the model loss function to assure model generalization through loss convergence, and detect the possible presence of vanishing gradient (Section 2.2.1.2).

- **Accuracy**: One of the most explanatory metrics is a model generalization, which can be evaluated in terms of classification precision. For both training and tests we plot the Top1 and Top5[2] accuracy curve, making sure they converge, with appropriate parameters. Training values, especially in contrastive learning, are reported as an indicator but do not represent the real accuracy. Besides that, we will only show the final test accuracies, which represent the actual model predicting capability.

- **Confusion Matrix**: Every test run is supported with prediction tracking on each class, with the help of a multiclass confusion matrix, represented as a $N \times N$ grid where N is the number of classes of target dataset $\mathbb{D}$. Each column indicates the model predictions of inputs, while rows

[2]In Top1 accuracy, an example is counted as correctly predicted if it is the first model prediction. In Top5 accuracy, the set of the first most probable output is taken instead, i.e. if the correct label lays in such set it is considered as correctly predicted.

represent the actual distribution of examples belonging to the same class. Based on this definition, we would expect a good classifier to have high values on the diagonal of its confusion matrix, while low ones on the other cells.

- **Matthews correlation coefficient (MCC)**: Information related to confusion matrix are also used to compute other evaluation metrics: Precision, Recall, F1 Score allows to give a more detailed overview on test performances. However, those metrics are especially effective when datasets are not class balanced, thus with different numbers of samples per class. Being CIFAR10 and CIFAR100 balanced, it turns out the outcomes are very similar to each other. We then decided to make use of the MCC evaluation, also called the *Phi coefficient*, which gives a more informative result than previously cited metrics [9]. Being c the total number of samples correctly predicted and s the test set cardinality, we define $\hat{t}$ as the number of true occurrences of each class and $\hat{p}$ as the number of predictions of samples within each class, then Matthews correlation coefficient is in the form:

$$MCC = \frac{cs - \hat{t} \cdot \hat{p}}{\sqrt{s^2 - \hat{p} \cdot \hat{p}}\sqrt{s^2 - \hat{t} \cdot \hat{t}}} \tag{5.2}$$

  multiclass MCC ranges in the interval [-1, +1], where -1 indicates complete disagreement between predictions and true classes, 0 no relationship, and +1 perfect agreement.

- **t-Distributed Stochastic Neighbourh Embedding (t-SNE)**: In order to express and evaluate feature representation and its distribution in the feature space, we plot t-SNE of different model layers. Particular attention is given to the last ResNet layer and the linear classification layer. t-SNE is an unsupervised algorithm that comes very helpful in data visualization thanks to its non-linear dimensionality reduction [58]. The criteria to evaluate similar data in the projected 2-dimensional space is the euclidean distance of points, computed on the original space. As we are going to see in the next section, we use t-SNE in conjunction to the Feature Extractor introduced in previous section to plot effective representations of models weights.

5.4 Experiments

As briefly explained in Section 5.2, we organized the experiments dividing the training into two parts: the first self-supervised (contrastive) learning with a generation of adversarial inputs, which followed fine-tuning. We decided to set a robust tuning, giving to the final linear layer adversarial images with ϵ ratio equal to the one applied to the correspondent contrastive model. In this scenario, we explored even further the effects of adversarial training, by comparing some robust models with no robust fine-tuning as well. The purpose was to find how much impact adversarial training has on the smaller problem of one-layer standard classification.

In Figure 5.1 we report an illustration of some natural inputs whose adversarial counterparts were misclassified by our model.

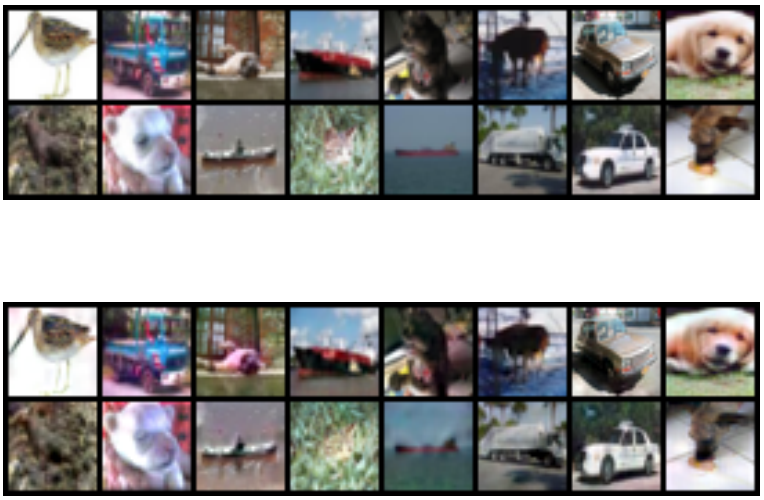


Figure 5.1: Representation of some misclassified inputs and the respective adversarial version. To clearly expose the effect of PGD we set $\epsilon = 5$.

Before moving on to the exposition of the experiments, we introduce a notation that will come in handy for the simplification and understanding of the results. We recall the use of adversarial training in order to make the classifier robust to all intents and purposes. According to this aspect, it is thus required to specify the ϵ value we apply to each model.

Definition. *Let $RoSS_{\mathbb{S}\to\mathbb{T}}$ (Robust Self-Supervised) be our fine-tuned self-supervised model. Then $\mathbb{S}$ represents the source classification task, while $\mathbb{T}$ the target classification task.*

5.4.1 Robust Contrastive Model Fine-Tuned on Same Predicting Task

The first experiment consists of a series of tests on different models. The unifying criteria is the use of the same classification task (i.e. the dataset used to train the network is the same when fine-tuning and testing). The notation defining which set has been utilized is composed of two names: the first one denotes the contrastive classification task, while the second the standard classification task.

To set up a proper range on adversarial noise, we ran the same classification problem with ϵ parameter in the range $[0, 1]$, with intervals of magnitude 0.2. Moreover, unlike we did in the training phase, we set PGD iterations to $K = 20$ (instead of 3) to highly increase the challenge for our networks. PGD step size remains $\sigma = 3$ instead. The result is 36 simulations, in which we track all the evaluation metrics described in Section 5.3.

It is clear from data collected how much impact the PGD method has on non-robust models: as we expected the higher we push ϵ on test inputs, the more the performance drops, which shows up with a large accuracy decrease for less robust models.

Accuracy $RoSS_{C10 \to C10}$						
Fine-Tuned	Evaluation ϵ Parameter					
Model ϵ	0	0.2	0.4	0.6	0.8	1
0	74.79	43.45	18.14	5.21	1.12	0.21
0.2	70.81	59.9	47.45	36.72	26.79	17.9
0.4	67.81	60.07	51.19	42.88	35.08	26.88
0.6	65.47	58.63	52.27	45.46	38.68	31.63
0.8	62.56	57.67	52.04	46.92	41.15	35.38
1	57.97	54.61	49.95	46.03	41.81	37.29

Table 5.1: Top1 test accuracy on CIFAR10, of contrastive model fine-tuned with standard classification on CIFAR10.

Observing carefully each column, we can also notice the limits on both non-robust and robust models. On natural inputs (evaluation $\epsilon = 0$), a robust classifier performs worse than a standard one, but the last appears completely prone to adversarial samples, and will totally fail to classify properly under

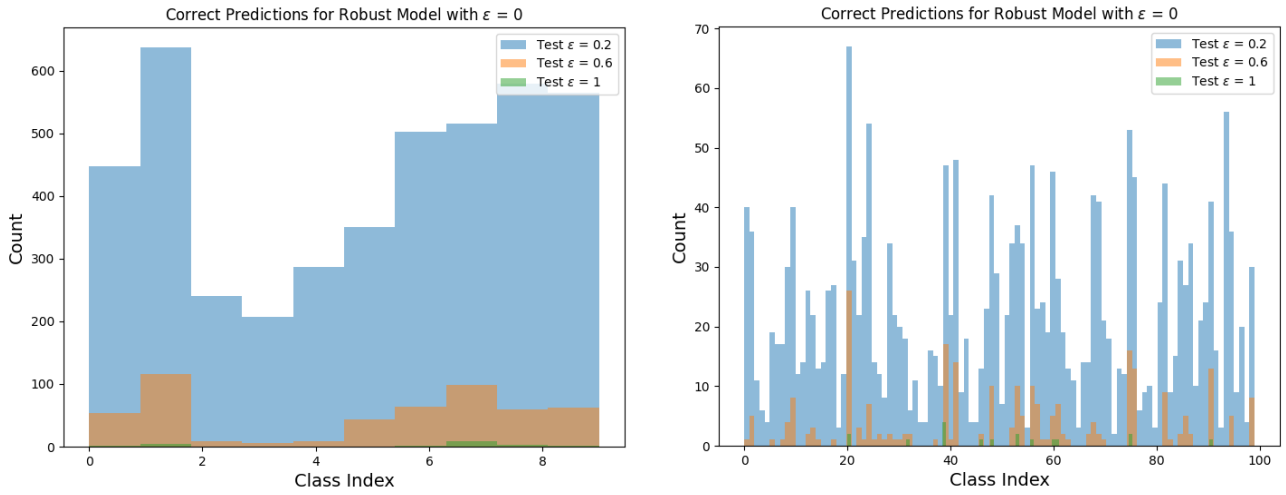


Figure 5.2: Comparison between correct predictions of $RoSS_{C10 \to C10}$ (on the left) and $RoSS_{C100 \to C100}$ (on the right). Standard models highly fail to correctly classify even with a small amount of PGD ϵ.

targeted attacks. Moreover, on the standard model, accuracy and MCC drops are consistent: an adversarial image with a PGD perturbation of magnitude $\epsilon = 0.2$ will halve the metrics for a non-robust model, especially on harder tasks, as CIFAR100 suggests.

Accuracy $RoSS_{C100 \to C100}$						
Fine-Tuned	Evaluation ϵ Parameter					
Model ϵ	0	0.2	0.4	0.6	0.8	1
0	46.51	21.84	7.61	2.58	0.67	0.17
0.2	43.89	35.11	27.39	20.87	14.79	10.22
0.4	41.66	35.73	29.81	24.27	19.62	15.34
0.6	39.02	34.5	30.0	25.33	21.37	17.76
0.8	36.67	33.0	29.41	25.9	22.6	19.35
1	33.17	30.33	27.36	24.53	21.82	19.33

Table 5.2: Top1 test accuracy on CIFAR100, of contrastive model fine-tuned with standard classification on CIFAR100.

On the other hand, using high ϵ values in the training phase leads to more consistent classifiers, even though on natural samples they lower its perfor-

mance (for $\epsilon = 1$, specifically of 16.82% on $RoSS_{C10 \to C10}$ and of 13.34% on $RoSS_{C100 \to C100}$).

In Figure 5.3 we compare the confusion matrices (CM) of the natural classifier and robust one on CIFAR10 (we did not show CIFAR100 for the obvious reason of grid space). We also point out the different scales between matrices in terms of colors. It is clear how some classes share common features by the fact all the CM have similar patterns on misclassified samples. Furthermore, when applying adversary inputs this phenomenon intensifies but does not change in terms of targets, which we would have expected since we only applied general PGD with no specific target for each attack.

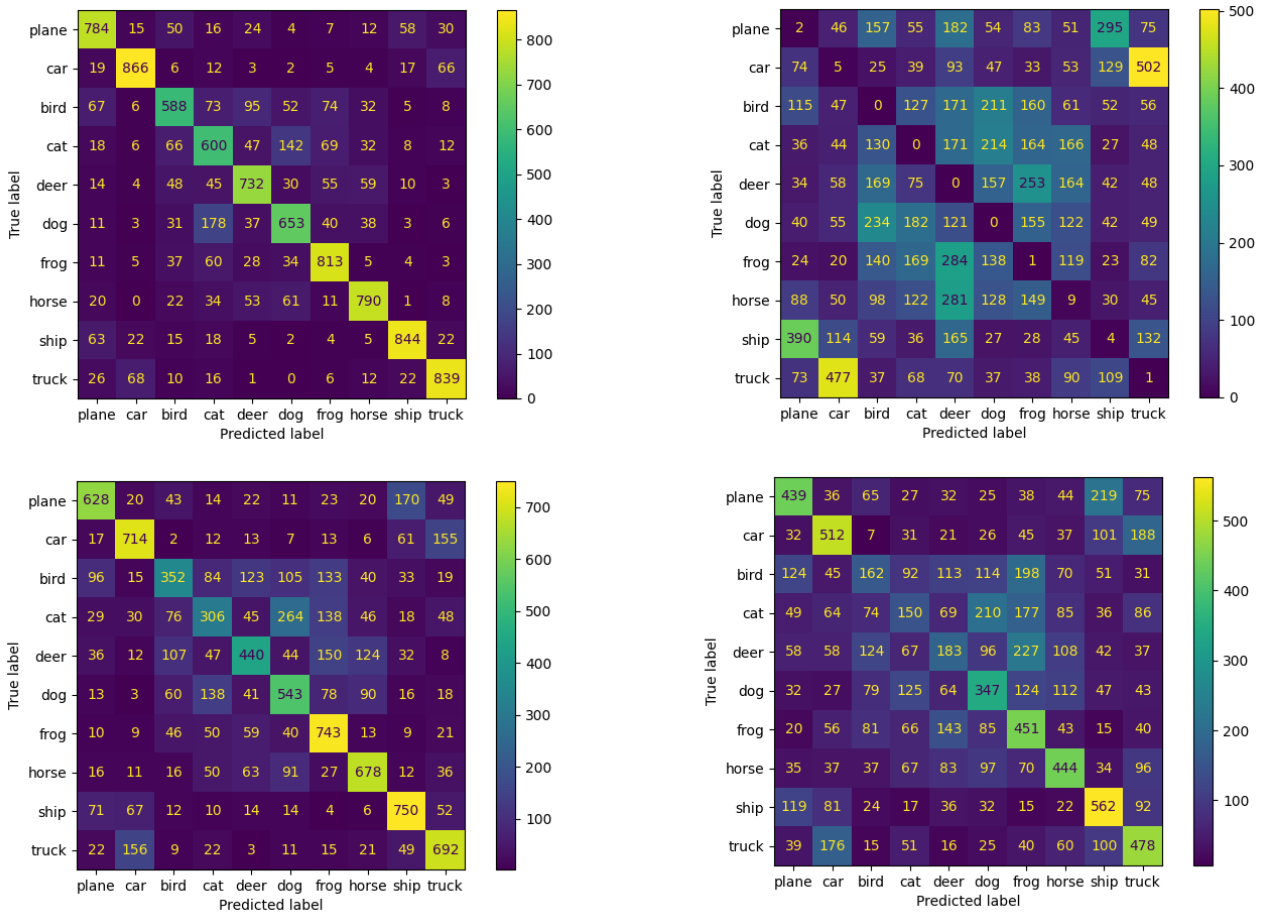


Figure 5.3: Comparing a standard $RoSS_{C10 \to C10}$ on the upper part ($\epsilon = 0$) and a robust $RoSS_{C10 \to C10}$ on the bottom ($\epsilon = 1$). Left represents a complete test on CIFAR10 with $\epsilon = 0$ while right with $\epsilon = 1$.

Let's finally consider the distribution of features in the last layer of our contrastive model, with and without adversarial learning applied to the training.

MCC $RoSS_{C100 \to C100}$						
Fine-Tuned	Evaluation ϵ Parameter					
Model ϵ	0	0.2	0.4	0.6	0.8	1
0	0.460	0.21	0.067	0.016	-0.003	-0.008
0.2	0.433	0.345	0.267	0.201	0.14	0.093
0.4	0.411	0.351	0.291	0.235	0.188	0.145
0.6	0.384	0.339	0.292	0.246	0.206	0.17
0.8	0.36	0.324	0.287	0.252	0.219	0.186
1	0.326	0.297	0.267	0.238	0.211	0.185

Table 5.3: MCC of SimCLR contrastive model trained on CIFAR100, then fine-tuned with standard classification on CIFAR100.

We show data with a t-SNE plot with target perplexity $p = 35$ and number of iterations `n_iter` $= 1500$. Results in Figure 5.4 show how robust parameters appear more sparse and wide in the feature space than the standard ones, again reinforcing the idea of adversarial training as a model generalization enhancer.

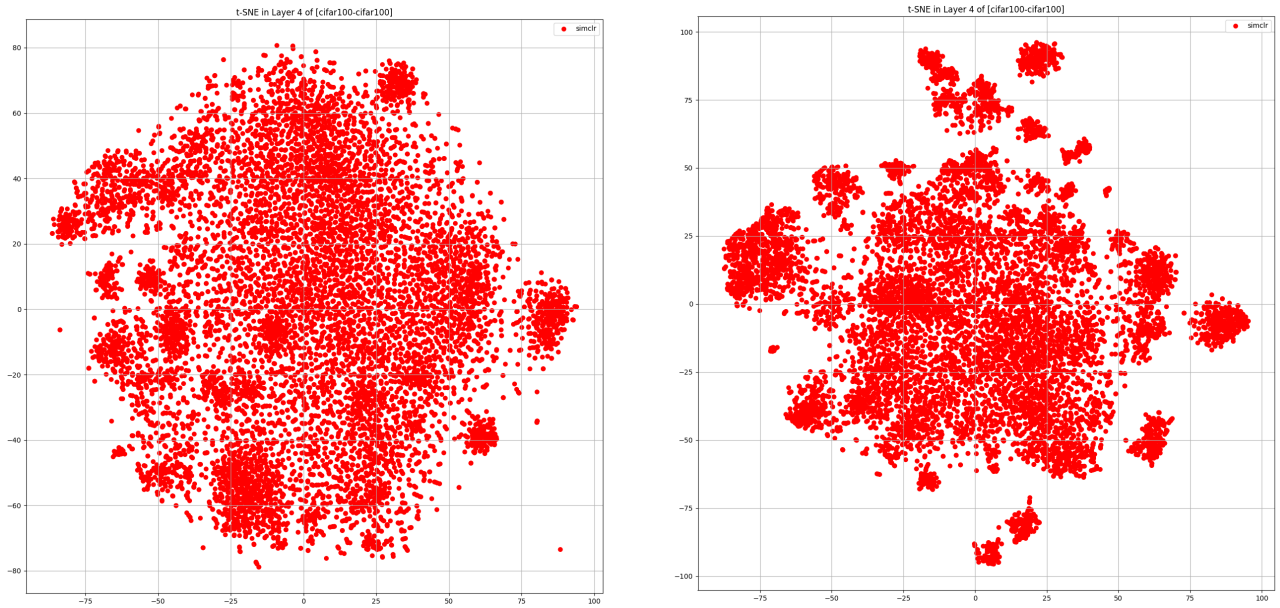


Figure 5.4: Feature representation of $RoSS_{C100 \to C100}$ trained with adversarial learning (right) and without (left). Features tend to spread more when trained robustly, but appear more agglomerate.

5.4.2 Robust Contrastive Model Fine-Tuned on Different Predicting Task

The previous experiment demonstrates how adversarial training makes a difference in preventing classifiers to fail when subjected to adversarial attacks or just to random adversarial samples. The results obtained in a self-supervised architecture reinforce the paradigm as better performing than simple supervision. We now proceed with a second experiment in which we test the architecture flexibility by changing classification tasks when fine-tuning the network. Since we aim to find every possible improvement from the based method, we first set the same parameters linked to adversarial training, according to the previous experiment. We still execute simulations for different ϵ values in the range $[0, 1]$ with intervals of 0.2, and using PGD method with iterations $K = 20$ and $\sigma = 3$. Finally, we set a test batch size of 1024, since we do not change any parameter. Accuracy results can be seen in Tables 5.4 and 5.5. Switching task results in a slight drop in performance, in fact, when stronger ϵ was applied, we assisted to less improvement than previous models: $RoSS_{C100 \to C10}$ robust accuracy tends to converge more than $RoSS_{C10 \to C10}$ when stronger PGD attacks are applied, which can be seen looking at the bottom-right area of Table 5.4, while $RoSS_{C10 \to C100}$ performs even worse when trained on $\epsilon = 1$ than $\epsilon = 0.8$, even though it was a trend manifesting in $RoSS_{C100 \to C100}$ too.

Accuracy $\boldsymbol{RoSS_{C100 \to C10}}$						
Fine-Tuned Model ϵ	Evaluation ϵ Parameter					
	0	0.2	0.4	0.6	0.8	1
0	69.51	36.15	12.68	3.07	0.58	0.04
0.2	65.11	55.04	44.01	33.18	23.25	15.92
0.4	62.6	55.2	47.08	39.19	32.04	24.89
0.6	59.51	54.04	48.06	42.05	36.25	30.3
0.8	57.27	52.25	47.36	42.73	37.76	33.15
1	53.94	49.96	45.98	42.12	38.18	34.09

Table 5.4: Top1 test accuracy on CIFAR10, of a contrastive model trained on CIFAR100 and then fine-tuned with standard classification on CIFAR10. Both pieces of training used the same ϵ value when adversarial learning was applied.

An interesting fact is that performances following a change of classification

task, are in general not so different from those in which the same task is maintained until the end of training. This result leads us in stating that contrastive learning does not limit performance in the same way fine-tuning does, which appears more crucial in defining the final model accuracy. This can be linked with the capacity of self-supervised models to adapt to different tasks, making use of better feature representations than the supervised counterparts.

We next show in Figure 5.5 confusion matrices comparison exclusively under adversarial attacks with $\epsilon = 1$. The matrices report a similar situation for both $RoSS_{C10 \to C10}$ and $RoSS_{C100 \to C10}$, with a small loss for the second one when trained robustly, which makes the two models behave almost identically.

Accuracy $\boldsymbol{RoSS_{C10 \to C100}}$						
Fine-Tuned	Evaluation ϵ Parameter					
Model ϵ	0	0.2	0.4	0.6	0.8	1
0	42.59	19.52	6.64	1.89	0.47	0.14
0.2	40.52	32.31	25.03	18.58	13.61	9.57
0.4	38.49	33.03	27.19	22.24	17.82	13.81
0.6	35.83	31.4	27.25	23.06	19.24	16.09
0.8	33.09	29.51	26.32	23.06	20.1	17.3
1	28.42	25.85	23.65	21.19	18.85	16.68

Table 5.5: Top1 test accuracy on CIFAR100, of a contrastive model trained on CIFAR10 and then fine-tuned with standard classification on CIFAR100. Both pieces of training used the same ϵ value when adversarial learning was applied.

To conclude the analysis on the CIFAR100 classification task, we also look to the prediction capacity of $RoSS_{C100 \to C100}$ in comparison with $RoSS_{C10 \to C100}$, illustrated in Figure 5.6. Prediction tracking detects similar results for each grade of robustness. In particular for increasing test ϵ values, the models start behaving more similar in both cases, as the small difference in blue and yellow/-green counts witnesses. This fact enforces what we previously stated, regarding contrastive learning being a flexible method for task generalization.

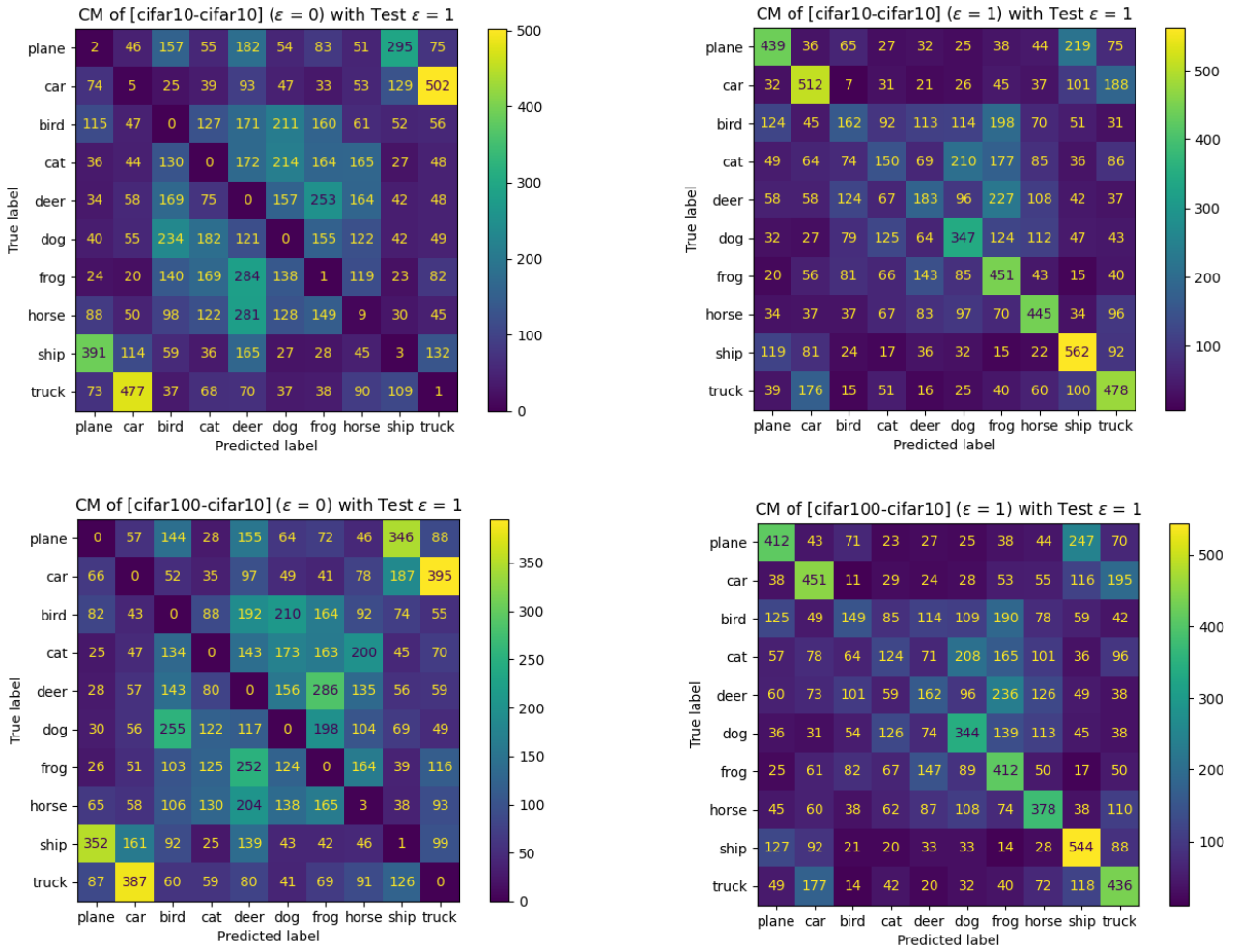


Figure 5.5: Comparing $RoSS_{C10 \to C10}$ on the upper part and $RoSS_{C100 \to C10}$ on the bottom (we now only compare on adversary examples: $\epsilon = 1$). Left shows non-robust versions of models and right the robust counterpart.

5.4.3 Effects of Adversarial Fine Tuning on Contrastive Model Robustness

The last experimental results demonstrate how self-supervised training fits well in learning effective feature representations. Using our contrastive models in different tasks does not affect consistently in resulting performances.

We now move on focusing on fine-tuning part of the training process. The aim is to understand how much robust fine-tuning is important to get correct predictions on critical inputs. We start by running some simulations similarly to what we made for the other experiments: this time we choose to group the results based on the test dataset, thus, grouping those classifiers that map their

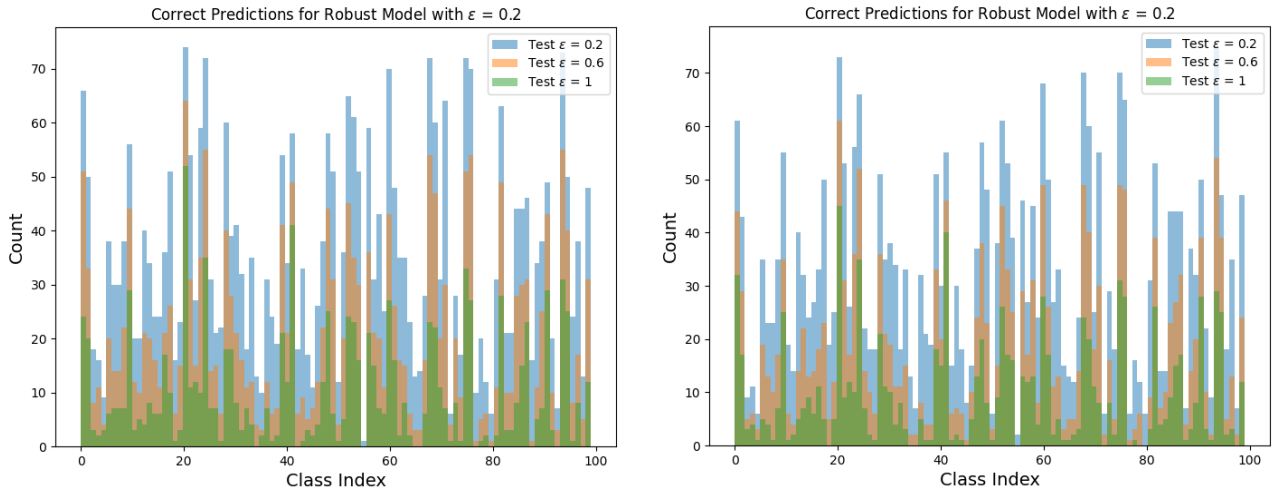


Figure 5.6: Comparison between correct predictions of $RoSS_{C100 \to C100}$ (on the left) and $RoSS_{C10 \to C100}$ (on the right). Cardinality appears very similar, especially on classes distribution, even though the contrastive classification task was different.

output in the same space. In other words, the comparisons are made on the model's test classification task.

We first execute a non-robust fine-tuning (setting PGD $\epsilon = 0$) for each robust contrastive model (each one previously trained with $\epsilon = 1$ on SimCLR). In this way, we get two different representations of the same classifier, where the difference, in terms of features, is on the final linear layer, trained with standard classification. Model testing on CIFAR10 makes the result evident: adversarial learning helps the model in gaining performance against possible attacks, as increasing accuracy is detected for adversarial inputs when ϵ rate gets higher. This phenomenon is best underlined by an 8.47% accuracy gain for $RoSS_{C100 \to C10}$. However, testing on CIFAR100 shows a different perspective: here accuracy does improve but by a smaller magnitude. In this last case, it may not be convenient to perform robust fine-tuning, since the risk is to lose in overall performance. That being said, the maximum absolute performance is 33.84% on average, which suggests that our architecture generalizes poorly CIFAR100.

Accuracy on CIFAR10						
Fine-Tuned Model	Evaluation ϵ Parameter					
	0	0.2	0.4	0.6	0.8	1
$RoSS_{C10 \to C10}$ $\epsilon = 1$	58.44	54.58	50.41	45.95	41.62	37.28
$RoSS_{C10 \to C10}$ $\epsilon = 0$	61.57	55.41	48.79	41.78	35.23	29.32
$RoSS_{C100 \to C10}$ $\epsilon = 1$	53.94	50.0	45.94	42.18	38.28	34.11
$RoSS_{C100 \to C10}$ $\epsilon = 0$	57.13	50.73	44.3	37.58	31.53	25.64
Robustness Improvement						
$RoSS_{C10 \to C10}$	-3,13	-0.83	1.62	4.17	6.39	7.96
$RoSS_{C100 \to C10}$	-3,19	-0.73	1.64	4.6	6.75	8.47

Table 5.6: Differences between adversarial and standard fine-tuning on the CIFAR10 classification task. Results show it is crucial to train adversarially even in standard classification to improve robustness.

Accuracy on CIFAR100						
Fine-Tuned Model	Evaluation ϵ Parameter					
	0	0.2	0.4	0.6	0.8	1
$RoSS_{C100 \to C100}$ $\epsilon = 1$	33.18	30.4	27.35	24.47	21.77	19.33
$RoSS_{C100 \to C100}$ $\epsilon = 0$	35.21	30.94	27.19	22.97	19.58	16.15
$RoSS_{C10 \to C100}$ $\epsilon = 1$	28.56	25.92	23.62	21.14	18.84	16.64
$RoSS_{C10 \to C100}$ $\epsilon = 0$	30.39	27.05	23.36	19.98	16.65	14.17
Robustness Improvement						
$RoSS_{C100 \to C100}$	-2.03	-0.54	0.16	1.5	2.19	3.18
$RoSS_{C10 \to C100}$	-1.83	-1.13	0.26	1.16	2.19	2.47

Table 5.7: Differences between adversarial and standard fine-tuning on the CIFAR100 classification task.

Tables 5.6 and 5.7 underline a trend that also came up in the experiment 5.4.1: certainly adversarial training works in its attempt to make features less susceptible to targeted attacks or noise in images, however, it comes with some losses. In fact, as we noted from the runs, when we apply PGD in training

inevitably the model becomes less adept at recognizing "clear" samples, i.e., those that are not adversarial. We recognize therefore that in the application of such a technique there is a tradeoff, between the maximum performance in conditions of optimal inputs, and the robustness that allows the model to commit, in general, fewer errors. The maximum expression of this phenomenon is shown in Table 5.1, where on natural samples we retrieved a maximum drop of 16.82% in accuracy when robustly trained, with a gain, on the other hand, of 37.08% on strong adversarial samples.

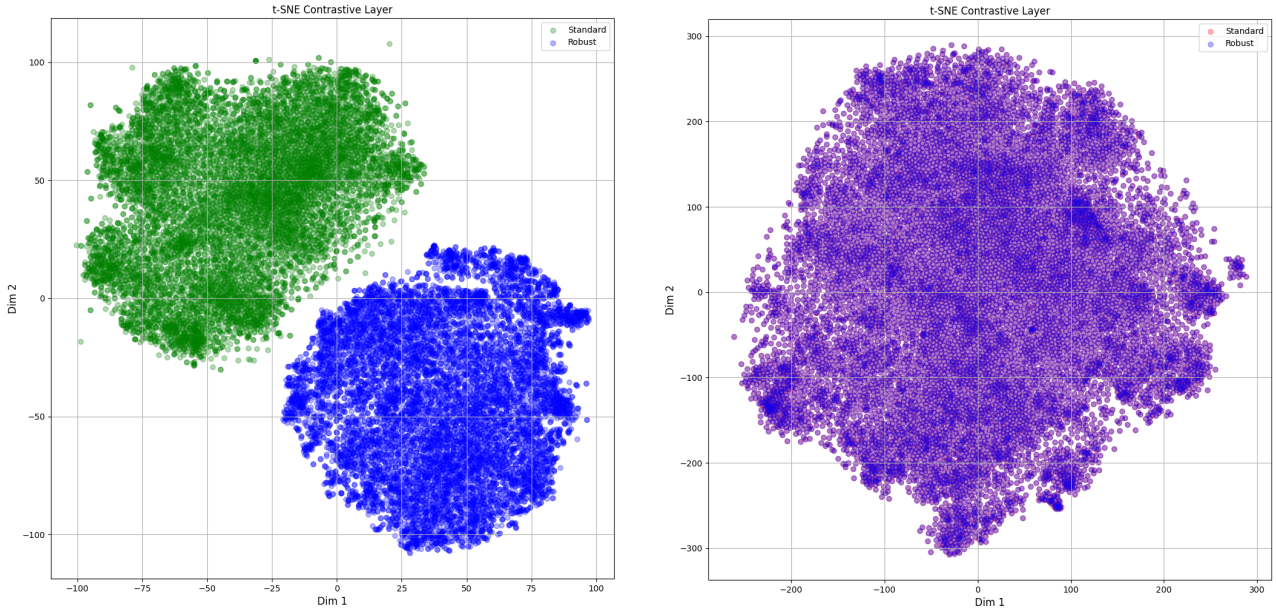


Figure 5.7: t-SNE feature extraction of the last $RoSS_{C10 \to C10}$ activation layer of SimCLR. The comparison is made between standard fine-tuning (left) and the robust one (right).

We then use the Feature Extractor to highlight weight changes between different trained models. As an example, we test feature variance loading pre-trained $RoSS_{C10 \to C10}$, with and without robust fine-tuning (as we best found, $\epsilon = 0$ and $\epsilon = 1$) and tracked the corrisponding features for a full standard model as well for comparison. The resulting t-SNE plots are illustrated in Figures 5.7 and 5.8. We utilized a perplexity $p = 50$ due to the high number of point samples, and set a number of iterations `n_iter = 2500`. What emerges, apart from the same distribution of features in the final activation of the contrastive layer, is the peculiar distribution in the final layer (on the partial-robust contrastive, feature are identical due to same robust learning before fine tutning, thus they overlap in purple points). While features of full-robust and full-

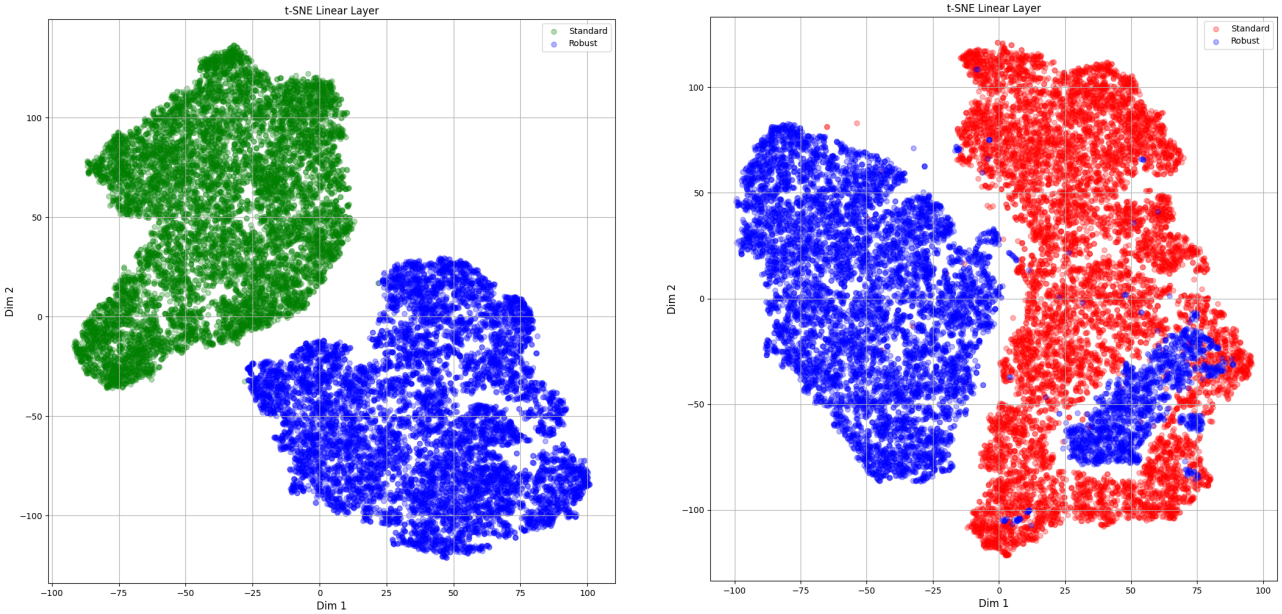


Figure 5.8: t-SNE feature extraction of the $RoSS_{C10 \to C10}$ linear layer. Colors represent the non-robust version (green), the completely robust (blue), and the partially robust (red).

standard models still appear well separated and more bounded, when changing fine-tuning adversarial criteria, some robust features fall off "clusterized" inside the standard feature set. It is reasonable to think that the blue stain inside the standard cluster suggests a property similarity between full-robust features and partial ones.

A different situation appears for partial robust $RoSS_{C100 \to C100}$, whose fine-tuning produces hybrid features that merge in the same cluster, as shown in Figure 5.10. This result may also be correlated with the low robust accuracy gain produced in Table 5.7.

Based on what is widely explained in Section 1.1.1, it is then necessary to conduct an analysis of the problem to evaluate the convenience of applying PGD or other methods to a designed model, especially if we want to submit a strong adversarial training. Anyway, in general, a model prone to adversarial attacks is not reliable for any application.

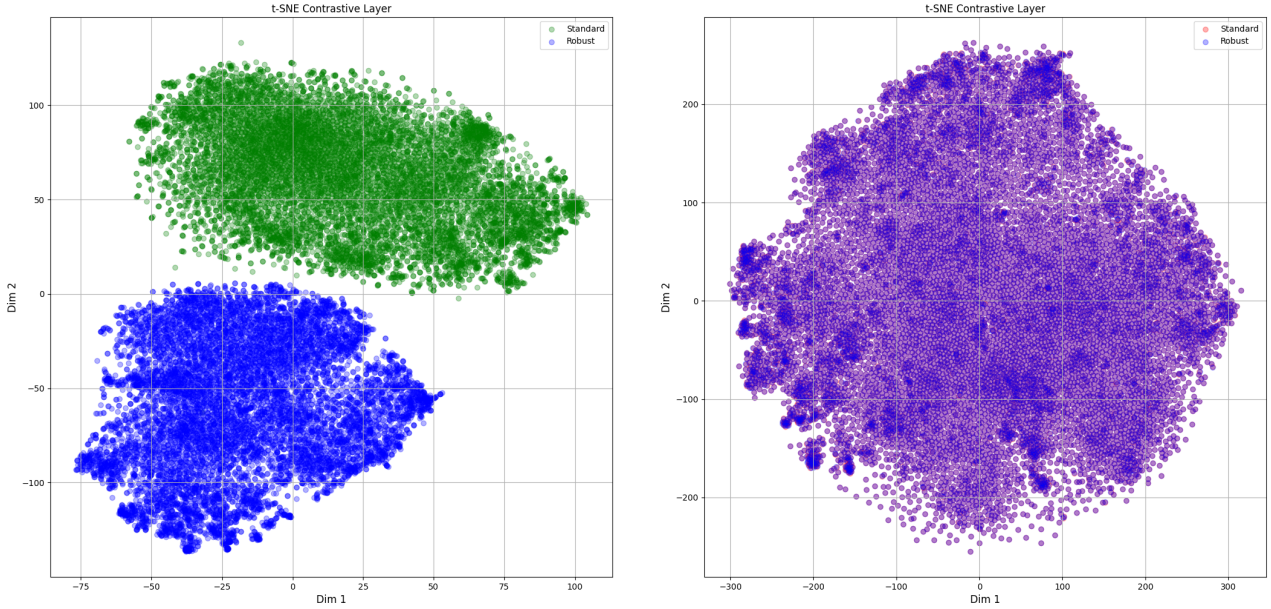


Figure 5.9: t-SNE feature extraction of the last $RoSS_{C100 \to C100}$ activation layer of SimCLR. The comparison is made between standard fine-tuning (left) and the robust one (right).

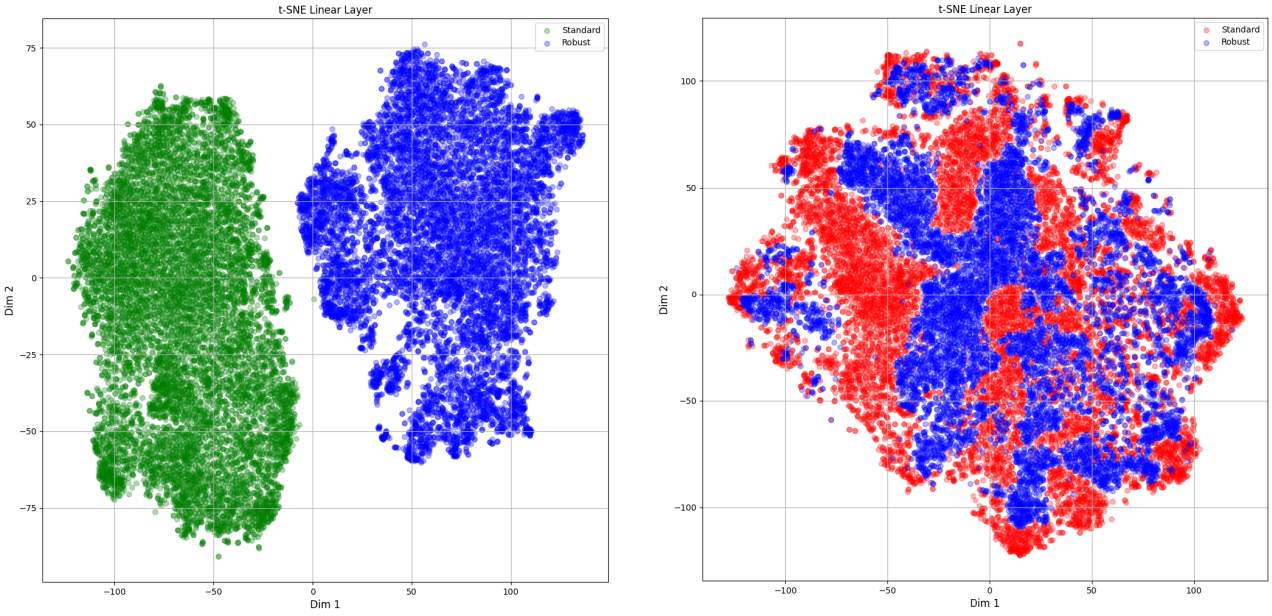


Figure 5.10: t-SNE feature extraction of the $RoSS_{C100 \to C100}$ linear layer. Colors represent the non-robust version (green), the completely robust (blue), and the partially robust (red).

Chapter 6

Conclusions

In the field of deep learning, the state of the art has made tremendous strides, including in the deepening of self-supervised learning techniques. As research advances, we have focused on the reliability of these models in terms of robustness. We reproduced SimCLR, a contrastive learning architecture, in a robust manner. Using Projected Gradient Descend (PGD) to perform Adversarial Training, we obtained different results, depending on the datasets used. The role of ϵ parameter turned out to be essential in order to visualize the effects of network improvement. Moreover, we made use of CIFAR10 and CIFAR100 datasets as support for our analysis. Changing datasets at different stages of the training framework allowed us to keep track of a wide range of simulations. We found that some challenging tasks had a negative impact on performances, as feeding CIFAR100 to our classifiers in fine-tuning step. On the other hand, different combinations, especially making use of the simpler CIFAR10 task, proved to work well since we retrieved satisfactory results.

That being said, the process of deep learning robustness improvement is still a wide object of research, and the state-of-the-art have not found the optimal solution in tacking adversaries detection yet. The problem might lie in the limiting premises, around which, deep learning problems are designed. Anyway this considerations goes beyond the purposes of the thesis, but represent an interesting point of view for future researches.

Bibliography

[1] S Abirami and P Chitra. "Energy-efficient edge based real-time healthcare support system". In: *Advances in Computers*. Vol. 117. 1. Elsevier, 2020, pp. 339–368.

[2] Md Zahangir Alom et al. *The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches*. 2018. arXiv: 1803.01164 [cs.CV].

[3] Ankesh Anand. *Contrastive Self-Supervised Learning*. https://ankeshanand.com/blog/2020/01/26/contrative-self-supervised-learning.html. 2020.

[4] Philip Bachman, R Devon Hjelm, and William Buchwalter. *Learning Representations by Maximizing Mutual Information Across Views*. 2019. arXiv: 1906.00910 [cs.LG].

[5] Pierre Baldi and Peter J Sadowski. "Understanding dropout". In: *Advances in neural information processing systems* 26 (2013), pp. 2814–2822.

[6] Christopher M Bishop. "Pattern recognition". In: *Machine learning* 128.9 (2006).

[7] Ting Chen et al. *A Simple Framework for Contrastive Learning of Visual Representations*. 2020. arXiv: 2002.05709 [cs.LG].

[8] Ting Chen et al. "Self-supervised generative adversarial networks". In: *arXiv preprint arXiv:1811.11212* 2 (2018).

[9] Davide Chicco and Giuseppe Jurman. "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation". In: *BMC Genomics* 21.1 (Jan. 2020), p. 6. ISSN: 1471-2164. DOI: 10.1186/s12864-019-6413-7.

[10] Anna Choromanska et al. *The Loss Surfaces of Multilayer Networks*. 2015. arXiv: 1412.0233 [cs.LG].

[11] Carl Doersch and Andrew Zisserman. *Multi-task Self-Supervised Visual Learning*. 2017. arXiv: 1708.07860 [cs.CV].

[12] Logan Engstrom et al. *Adversarial Robustness as a Prior for Learned Representations*. 2019. arXiv: 1906.00945 [stat.ML].

[13] Kunihiko Fukushima and Sei Miyake. "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition". In: *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[15] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014).

[16] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2015. arXiv: 1412.6572 [stat.ML].

[17] John A Hartigan and Manchek A Wong. "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the royal statistical society. series c (applied statistics)* 28.1 (1979), pp. 100–108.

[18] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[19] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[20] Kaiming He et al. *Momentum Contrast for Unsupervised Visual Representation Learning*. 2020. arXiv: 1911.05722 [cs.CV].

[21] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[22] R Devon Hjelm et al. *Learning deep representations by mutual information estimation and maximization*. 2019. arXiv: 1808.06670 [stat.ML].

[23] Hossein Hosseini et al. *On the Limitation of Convolutional Neural Networks in Recognizing Negative Images*. 2017. arXiv: 1703.06857 [cs.CV].

[24] AG Ivakhnenko. *Cybernetic systems with combined control*. Tech. rep. FOREIGN TECHNOLOGY DIV WRIGHT-PATTERSON AFB OHIO, 1967.

[25] Alexey Grigorevich Ivakhnenko. "Polynomial theory of complex systems". In: *IEEE transactions on Systems, Man, and Cybernetics* 4 (1971), pp. 364–378.

[26] Rohini Khalkar, Adarsh Dikhit, and Anirudh Goel. "Handwritten Text Recognition using Deep Learning (CNN & RNN)". In: *IARJSET* 8 (June 2021), pp. 870–881. DOI: 10.17148/IARJSET.2021.86148.

[27] Günter Klambauer et al. *Self-Normalizing Neural Networks*. 2017. arXiv: 1706.02515 [cs.LG].

[28] *Know your enemy: How you can create and defend against adversarial attacks*. https://towardsdatascience.com/know-your-enemy-7f7c5038bdf3. Accessed: 21/02/2022.

[29] Alexander Kolesnikov, Xiaohua Zhai, and Lucas Beyer. *Revisiting Self-Supervised Visual Representation Learning*. 2019. arXiv: 1901.09005 [cs.CV].

[30] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. *Adversarial Machine Learning at Scale*. 2017. arXiv: 1611.01236 [cs.CV].

[31] Yann LeCun et al. "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

[32] Mohammed Quddus Lipika Deka. "Network-level accident-mapping: Distance based pattern matching using artificial neural network." In: *National Center for Biotechnology Information* (December 2013). DOI: DOI: 10.1016/j.aap.2013.12.001.

[33] Xiao Liu et al. "Self-supervised Learning: Generative or Contrastive". In: *IEEE Transactions on Knowledge and Data Engineering* (2021), pp. 1–1. ISSN: 2326-3865. DOI: 10.1109/tkde.2021.3090866.

[34] Aleksander Madry et al. *Towards Deep Learning Models Resistant to Adversarial Attacks*. 2019. arXiv: 1706.06083 [stat.ML].

[35] Sonali B Maind, Priyanka Wankar, et al. "Research paper on basic of artificial neural network". In: *International Journal on Recent and Innovation Trends in Computing and Communication* 2.1 (2014), pp. 96–100.

[36] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[37] Manjula R. Muni Kumar N. "Design of Multi-layer Perceptron for the Diagnosis of Diabetes Mellitus Using Keras in Deep Learning." In: *Satapathy S., Bhateja V., Das S. (eds) Smart Intelligent Computing and Applications. Smart Innovation, Systems and Technologies* 104 (2019). DOI: https://doi.org/10.1007/978-981-13-1921-1_68.

[38] Suvda Myagmar, Adam J Lee, and William Yurcik. "Threat modeling as a basis for security requirements". In: *Symposium on requirements engineering for information security (SREIS)*. Vol. 2005. Citeseer. 2005, pp. 1–8.

[39] In Jae Myung. "Tutorial on maximum likelihood estimation". In: *Journal of mathematical Psychology* 47.1 (2003), pp. 90–100.

[40] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Icml*. 2010.

[41] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018. arXiv: 1811.03378 [cs.LG].

[42] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. *Representation Learning with Contrastive Predictive Coding*. 2019. arXiv: 1807.03748 [cs.LG].

[43] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural networks* 12.1 (1999), pp. 145–151.

[44] Herbert Robbins and Sutton Monro. "A stochastic approximation method". In: *The annals of mathematical statistics* (1951), pp. 400–407.

[45] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[46] Virginia R de Sa. "Learning classification with unlabeled data". In: *Advances in neural information processing systems*. Citeseer. 1994, pp. 112–119.

[47] Mike Schuster and Kuldip K Paliwal. "Bidirectional recurrent neural networks". In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[48] Federico Sergio. *Robustness Analysis of Self-Supervised Models*. Version 2.0.4. Mar. 2022.

[49] Anish Shah et al. "Deep Residual Networks with Exponential Linear Unit". In: *Proceedings of the Third International Symposium on Computer Vision and the Internet* (Sept. 2016). DOI: 10.1145/2983402.2983406.

[50] Carl-Johann Simon-Gabriel et al. *First-order Adversarial Vulnerability of Neural Networks and Input Dimension*. 2019. arXiv: 1802.01421 [stat.ML].

[51] Samuel L Smith et al. "Don't decay the learning rate, increase the batch size". In: *arXiv preprint arXiv:1711.00489* (2017).

[52] Kihyuk Sohn et al. *FixMatch: Simplifying Semi-Supervised Learning with Consistency and Confidence*. 2020. arXiv: 2001.07685 [cs.LG].

[53] Bert Speelpenning. "Compiling fast partial derivatives of functions given by algorithms". PhD thesis. University of Illinois at Urbana-Champaign, 1980.

[54] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[55] Richard Sutton. "Two problems with back propagation and other steepest descent learning procedures for networks". In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*. 1986, pp. 823–832.

[56] *Una guida pratica a ReLU*. https://ichi.pro/it/una-guida-pratica-a-relu-202570656444919. Accessed: 21/02/2022.

[57] *Understanding Contrastive Learning: Learn how to learn without labels using self-supervised learning*. https://towardsdatascience.com/understanding-contrastive-learning-d5b19fd96607. Accessed: 21/02/2022.

[58] Laurens Van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE." In: *Journal of machine learning research* 9.11 (2008).

[59] Svante Wold, Kim Esbensen, and Paul Geladi. "Principal component analysis". In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52.

[60] Qizhe Xie et al. *Unsupervised Data Augmentation for Consistency Training*. 2020. arXiv: 1904.12848 [cs.LG].

[61] Dong Yu and Li Deng. "Deep learning and its applications to signal and information processing [exploratory dsp]". In: *IEEE Signal Processing Magazine* 28.1 (2010), pp. 145–154.