POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Thesis

Adversarial Attacks for Convolutional Neural Networks and Capsule Networks

Supervisors

Prof. Maurizio MARTINA

Prof. Muhammad SHAFIQUE

Prof. Andreas STEININGER

Project Ass. Alberto MARCHISIO

Candidate

Giovanni CARAMIA

Academic Year 2020/2021

Acknowledgements

First of all, I would to thank Professor Maurizio Martina for allowing me to explore a new field of research. I am grateful to him for giving me the chance to interact and develop the thesis in collaboration with the TU Wien and NYU Abu Dhabi. In this regard, I would to thank the Professor Andreas Steininger and Professor Muhammad Shafique for this opportunity. A special thanks goes to my tutor Alberto, a kind and very competent person who took me by the hand and taught me a lot of things about this new field: I also thank him for the help and support throughout the development of the thesis .
Moreover, I would also like to thank my family, my father Gino and my mother Enza, without whom, all this would not have been possible. They have always encouraged me to try to improve myself and always go one step further.
Furthermore, I would like to thank my friends, which is like a second family. Thanks to them, and to our moments of lightheartedness, everything was more bearable. In particular, I would like to thank Enrico, Lele, Elena, Annette and Marco for the philosophical chats under the Mole. I would like to thank Adalgisa, Carmela, Cristina and Alessandra for supporting me. I cannot fail to mention my historical friends: Gianfranco, Pierluigi, Gigi, Nicola, Martino, Ramon e Peppino for the lovely conversations.

"To my grandmother Giuditta"

"A man should look for what is,
and not for what he thinks should be."
Albert Einstein

Summary

In Computer Vision (CV) context, the image classification is a supervised learning problem which has several applications in all fields such as autonomous driving, medical diagnosing, remote sensing and so on. In the Deep Learning (DL) field, the image classification problem is solved by several architectures: from the simple Convolutional Neural Networks (CNNs) to the more complex models such as Capsule Networks (CapsNets). Nowdays, it is complicated to apply the CapsNet to complex and high resolution datasets, and consequently this leads to an increase in the complexity of the CapsNet architecture. Moreover, CapsNets for some datasets are still unexplored. An important aspect of image classification is the robustness of the architectures against adversarial attacks: an image can be misclassified crafting a small perturbation to the input. In this dissertation, the adversarial attacks are crafted on Residual Neural Networks (ResNet) and CapsNet models. In a CapsNet configuration, another way to inject attacks is done through Vote Attack, which directly attacks the votes instead of output capsules. In the literature, until now the impact of the Vote Attack has been evaluated only on simple datasets.
Moreover, in the classification of outdoor images, the effect of atmospheric phenomena has not yet been analyzed and could affect security. In this context, in the field of autonomous driving, a misclassification could generate very serious effects such as road accidents and dangers for passengers and pedestrians.
To resume, the thesis focus on these topics and tries to solve some challenges:

i. *What is the behaviour of the CapsNet on complex and high resolution datasets? In this regard, what about the robustness of the CapsNets against adversarial attacks?*

ii. *What is the impact of Vote Attack on complex datasets?*

iii. *How is it possible to reproduce the effect of atmospheric conditions on the camera lens in autonomous driving?*

The **research contributions of the thesis** are mainly two:

- Evaluate the performances and the robustness of CapsNet models against adversarial attacks

- Design an adversarial attack that can mislead the CNNs and CapsNet architectures

Going into details, in the first part of this dissertation there is a systematic analysis of the **latest** CapsNet architectures in literature (ShallowCaps, DeepCaps, YaoCaps and FMCaps) applied to the most common benchmarks datasets (SVHN, CINIC10, CIFAR10, CIFAR100). Furthermore, CapsNets are applied **for the first time** on high resolution datasets (MLRS and COVID19Ti). In particular, the COVID19 Ti dataset contains several Chest X-ray images used for medical diagnosing decision. It is splitted into two datasets: COVID19-CXR (grey-scale configuration) and COVID19-ENH (Chest X-ray images enhanced using image processing technique).Moreover, MLRS dataset is a novel dataset used for remote sensing problem: it contains high-resolution optical satellite images allowing to capture different perspectives of the Earth. Having high resolution datasets involves some implementation problems which affect the memory usage. One solution would be to resize the images to a smaller size: for example from 256x256 to 32x32. Some tricks have been applied to obtain good performances. For example, taking into consideration the case of ShallowCaps: in MLRS the kernel_size of the PrimaryCaps has been increased; in the case of COVID19 CXR, a Batch Normalization Layer was added after Conv2D and the spitting rate was changed from 85/15 to 50/50.
Furthermore, in the second part of thesis the robustness of CNNs and CapsNet models against adversarial attacks is evaluated. At the beginning, the robustness evaluation is performed on MLRS and COVID19Ti: the robustness is performed also changing the pnorm (l_∞ and l_2) for Remote sensing dataset. After that, the Vote Attack impact is evaluated on CIFAR100.

In this context, taking into account the accuracy as a measure of performances and the robust accuracy as a metric in the robustness field, **the results show that**:

- For all cases, DeepCaps performs better than ShallowCaps. In particular, considering SVHN and CINIC10 datasets, DeepCaps reaches an accuracy of 96.48% and 75.24%, respectively, while ShallowCaps gets 94.96% and 56.72%, respectively.

- Yao Caps achieves the best results for CIFAR10 (94.70%) and CIFAR100 (70.14%).

- DeepCaps performs better than ResNet models on high resolution datasets (MLRS and COVID19Ti), as shown in Table 1.

	COVID19 CXR	COVID19 ENH	MLRS
ResNet50	90.94 %	93.54 %	75.22 %
DeepCaps	**93.34** %	**94.01** %	**82.23** %

Table 1: Accuracy values of DeepCaps and ResNet50 on high resolution datasets

- DeepCaps is more robust than ResNets for small perturbations on high resolution datasets (MLRS and COVID19Ti): Table 2 shows that for perturbations greater than a certain ϵ the situation is reversed.

	COVID19 CXR	COVID19 ENH	MLRS (l_∞)
FGSM	$\epsilon > 0.013$	$\epsilon > 0.015$	/
PGD	$\epsilon > 0.017$	$\epsilon > 0.03$	$\epsilon > 0.03$
BIM	$\epsilon > 0.016$	$\epsilon > 0.025$	$\epsilon > 0.02$
MIM	$\epsilon > 0.018$	$\epsilon > 0.03$	$\epsilon > 0.025$

Table 2: For each attack the epsilon value is shown after which the situation is reversed (ResNet becomes more robust than DeepCaps)

  Furthermore, in the context of MLRS case, adversarial attacks with l_∞ norm behave better than those with l_2. As a consequence, the l_∞ allows the attack to be stronger.

- Vote Attack continues to be stronger than Caps Attack even for more complex datasets like CIFAR 100 because robust accuracy values due to Vote Attack are always lower than those caused by Caps Attack. For example, when $\epsilon = 0.04$, in **Vote** case the robust accuracies, i.e., the accuracies measured when at the input are applied the adversarial examples, of FGSM (11.27 %), PGD (0.17 %) and BIM (0.5 %) are less than FGSM (11.59 %), PGD (0.19 %) and BIM (0.63 %) in **Caps Attack**.

Moreover, in the real world there are several situations where DL models fool the correct classification due to **atmospheric phenomena**. In this context, a **novel methodology** is proposed: assuming that the camera lens is dirty due atmospheric conditions (such as rain, snow and hail), it is possible to craft a perturbation changing the pixels of the input following several patterns. Such attack, called **Pattern Attack**, is performed in a black box setting: the only information available is the probability associated to each output label. Going into depth, in the closest related works some researchers have perturbed the image by changing one or more pixels, according to several algorithms; others have added stickers to the images misleading the models. The **innovative idea** of the our

attack methodology is to modify the pixels, following some patterns that simulate the patterns that the drops or the snowflakes create when they are placed on the camera lens. In reality, a drop of water has a spherical shape while a snowflake has a hexagonal one. For simplicity, a drop or a snowflake can be modelled as a pixel: a small square of the image which has $h \cdot l$ pixels (h = height , l = length).
Pattern Attack is splitted into three attacks: **rain attack**, **snow attack** and **hail attack**. After crafting attack, the adversarial image goes into the black box in a way that there is a comparison between the **prior confidence** in a clean situation and the **new confidence** in an adversarial situation in order to evaluate the **attack success Rate** (ASR). Pattern Attack is applied to CIFAR10 and it is evaluated on three models: LeNet, ResNet and CapsNet. Figure 1 shows some examples of misclassification.

The results obtained by applying the Pattern Attack **show that**:

- All types of Pattern Attack work well both in LeNet and ResNet models. (ASR $>$ 65 %)
- Hail Attack is the best attack because it performs better than other attacks in all cases: LeNet ($ASR = 82{,}5\%$) , ResNet ($ASR = 78{,}5\%$) and CapsNet ($ASR = 63\%$)

Figure 1: Examples of misclassification for each Pattern Attack: it is possible to see that the model was deceived, the predicted class is different from the real one

The purpose of this work is to make an important contribution to the field of security. In Deep Learning context, CNNs and CapsNets will still be used in many tasks and it is necessary to try to improve their performances and robustness through new kinds of defense in order to find countermeasures against several adversarial attacks that are created every day. The steps of future research may be different: for the first part of analysis, the CapsNet models will be applied to more complex datasets such as ImageNet or ObjectNet trying to obtain good

classification performances. Moreover, in the context of Pattern Attack, the future developments consist of applying the attack on other higher resolution datasets and on other CapsNet models such as DeepCaps, YaoCaps and FMCaps.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

ASR
Attack Success Rate

BIM
Basic Iterative Method

BN
Batch normalization

CapsNet
Capsule Network

CNN
Convolutional Neural Network

COVID-19
COronaVIrus Disease 19

CV
Computer Vision

DDN
Decoupling Direction and Norm

DL
Deep Learning

DNN

Deep Neural Network

DR

Dynamic Routing

FC

Fully connected

FGSM

Fast Gradient Sign Method

GD

Gradient Descent

GPU

Graphics Processing Unit

ILSVRC

ImageNet Large Scale Visual Recognition Challenge

MIM

Momentum Iterative Method

ML

Machine Learning

MLP

Multilayer Perceptron

MSE

Mean Square Error

NN

Neural Network

PGD

Projected Gradient Descent

ResNet

Residual Neural Network

RGB

R=Red, G=Green, B=Blue

RL

Reinforcement Learning

SGD

Stochastic Gradient Descent

SL

Supervised Learning

SOTA

State Of The Art

UL

Unsupervised Learning

chapterpart

Part I

Computer Vision: Introduction

Chapter 1

Introduction

In Computer Vision (CV) context, the image classification is a supervised learning problem which has several applications in all fields such as autonomous driving, medical diagnosing, remote sensing and so on. In the Deep Learning (DL) field, the image classification problem is solved by several architectures: from the simple Convolutional Neural Networks (CNNs) to the more complex models such as Capsule Networks (CapsNets). An important aspect of image classification is the robustness of the architectures against adversarial attacks [1] : an image can be misclassified crafting a small perturbation to the input. [2] [3] In this dissertation, the adversarial attacks are crafted on Residual Neural Networks (ResNet) and CapsNet models. Over the years, many researchers have been interested in the robustness of Capsule Networks.[4] [5] [6] [7] [8] In a CapsNet configuration, another way to inject attacks is done through Vote Attack, which directly attacks the votes instead of output capsules.

1.1 Scientific Challenges

Nowdays, it is complicated to apply the CapsNet to complex and high resolution datasets, and consequently this leads to an increase in the complexity of the CapsNet architecture. Moreover, CapsNets for some datasets are still unexplored. In this context, Medical images and Remote sensing dataset will be evaluated. Moreover, it is necessary to understand the robustness of CapsNet for these new datasets. For example, in medical applications a misclassification can mean a wrong diagnosis instead of a correct one. Furthermore, in the CapsNet context, the Vote Attack is a recent attack and it has been evaluated only on simple datasets until now. It is important to try to see the impact of Vote Attack on more complex datasets.

Moreover, in the classification of outdoor images, the effect of atmospheric phenomena has not yet been analyzed and could affect security. In this context, in the field of autonomous driving, a misclassification could generate very serious effects such as road accidents and dangers for passengers and pedestrians. To resume, the thesis focus on these topics and tries to solve some challenges:

- *What is the behaviour of the CapsNet on complex and high resolution datasets? In this regard, what about the robustness against adversarial attacks?*
- *What is the impact of Vote Attack on complex datasets?*
- *How is it possible to reproduce the effect of atmospheric conditions on the camera lens in autonomous driving?*

1.2 Our Novel Contributions

The purpose of this dissertation is to solve these challenges. It is possible to highlight mainly three novel contributions:

1. Systematic analysis of recent CapsNets architectures for different datasets (chapters 3, 4, 5), in which they are evaluated for the first time on high resolution datasets.
2. Analysis of different adversarial attacks on CNNs and CapsNets, and comparison between CapsAttacks and Vote Attacks on complex datasets.
3. Pattern Attack design, implementation and evaluation.

1.3 Overview

The thesis is splitted into the following chapters :

- **Chapter 2:** Review of the basic notions of Neural Networks (NNs), Convolutional Neural Networks (CNNs), Residual Neural Networks (ResNet) and Adversarial Attacks
- **Chapter 3:** Review of the latest relevant Capsule Networks (CapsNet) architectures in literature with a systematic approach (ShallowCaps, DeepCaps, YaoCaps, FMCaps)
- **Chapter 4:** Evaluate the performances of CapsNet models on benchmark datasets (SVHN, CINIC10, CIFAR10, CIFAR100).

- **Chapter 5:** Evaluate the performances of CapsNet on high resolution datasets : MLRS and COVID19.
- **Chapter 6:** Review of the most common adversarial attacks in a white-box setting: FGSM, PGD, BIM and MIM
- **Chapter 7:** Compare the CapsNet robustness against adversarial attacks in a white box configuration for novel datasets and evaluate the impact of Vote Attack on on a complex dataset like CIFAR100
- **Chapter 8:** Design a novel methodology, called Pattern Attack, in a black box configuration: this attack is inspired to the real scenarios configuration
- **Chapter 9:** Summarize all the work with some future research developments

Chapter 2

Background

In Chapter 2 there will be touched some important basic concepts related to the Computer vision field. In particular, after a brief introduction about the image classification, Neural Networks will be introduced highlighting the basic concepts related. Going into depth, it is important to analyze two architectures that are fundamentals in deep learning fields : Convolutional Neural Networks (CNN) and Residual Neural Networks (ResNet). At the end, it is crucial dealing with the robustness of deep learning models, paying attention to what concerns the adversarial attacks.

2.1 Image Classification

In Machine Learning we have three kind of learning methods [9]:

i. **Supervised learning**: an human observer labels the images and groups them into a training dataset. Typical examples of SL are **classification** and **regression**.

ii. **Unsupervised learning**: the training dataset is **unstructured** and **unlabeled**. Typical example of UL is **Clustering**.

iii. **Reinforcement learning**: the system reacts to changes in its environment by learning from mistakes in order to improve its performance.

The **image classification** belongs to **supervised learning**. It takes into account that the input image has a specific shape (ex.32x32) and a certain colour(ex.gray-scale or RGB). The input volume is fed into the model which applies a certain algorithm so that the image classification is performed: the output declares which class the image belongs to. In this way it is possible to solve a classification problem.

2.2 Neural Networks: basic concepts

In this context, the Neural Networks (NNs) are widely spread. In particular, it is important to focus our attention on Multilayer perceptrons(MLP).

2.2.1 Multilayer perceptrons (MLP)

First of all, the best way to introduce the MLP is to focus on its main computational unit: the **perceptron**.

Perceptron

A perceptron is the simplest neural network: it consists a single neuron. In 1958 Rosenblatt [10] introduced this concept making an analogy between biological and artificial neurons.

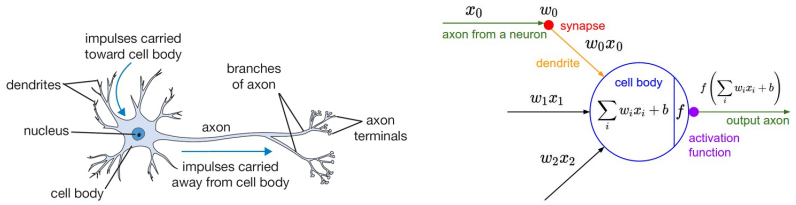


Figure 2.1: The neuron [11]

In human brain the biological neuron receives the information from the dendrites, and after some operations, compute the output to the synapses which communicates with another neuron. The artificial neuron works in a similar way: the inputs (dentrites) are fed into a function $f(x)$ (making some processes) which compute the output. Figure 2.1 shows the difference between a biological and an artificial neuron.

Multilayer perceptrons

A perceptron is not sufficient when the tasks begin more complex. The best idea is to create another model (MLP) that combines several neurons stacked in many layers, as shown in Figure 2.2. The basic concepts are the same as in perceptron configuration.

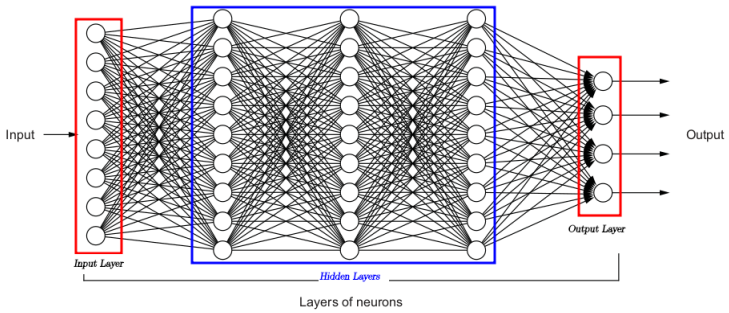


Figure 2.2: Multilayer perceptron architecture [12]

2.2.2 The learning process

After a brief introduction of the MLP model (a combination of several perceptrons), it is crucial to analyse the way of learning. Indeed, the learning process is a fundamental methodology which helps the neural network to learn. It is based on the repetition of three steps: i)feedforward calculations, ii) error computation, iii) backpropagation .

1. Feedforward process

The feedforward process is the method that allows to compute the output: the information flows from the input layer, to the hidden layers and at the end to the output layer. It is based on two consecutive steps:

- Weighted sum:

$$z = \sum x_i \cdot w_i + b \tag{2.1}$$

  Each neurons performs a dot product with the input $\mathbf{x}_i$ and its weights $\mathbf{w}_i$ and adds the bias **b**.

- Activation function:

$$\hat{y} = a = f(z) = f(\sum x_i \cdot w_i + b) \tag{2.2}$$

  In the second stage there is an application of the activation function which introduces the **non linearity** to the neural network: this is a gold point that gives us the possibility to solve any non-linear problems. [1]

[1]The most simple activation function is a step function that produces a binary output (0 or 1) : if z is bigger or equal to 0, the output must be 0; if z is less than 0, the output must be 1. In the design process it is possible to choose several activation functions, such as sigmoid function, softmax function, hyperbolic function, ReLu function and so on. **ReLu function** is the best choice in the hidden layers, while the **softmax fuction** is for output layer.

The perceptron learns step by step using a trial and error procedure. The neuron uses their weights tuning their values up and down learning from their mistakes, as a human do with experience. Following this procedure, the network makes a prediction $\hat{y}$.

2. Error computation

After that, it is crucial to evaluate the performance of the prediction: the error e quantifies if the network's prediction $\hat{y}$ is close to the true label y.

$$e = \hat{y} - y \tag{2.3}$$

Loss functions

The designer decides what is the error function, or cost function or loss function which fits better in that situation. In Table 2.1 there is an introduction of the most useful loss functions. [2]

MSE / L2 Loss / Quadratic Loss	$E(W,b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$
Cross Entropy	$E(W,b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\hat{y}_{i,j}\log(p_{i,j})$

Table 2.1: $E(W,b)$ is the loss function (also annotated as $J(W,b)$), W is the Weights matrix, b is the biases vector, N is the number of training examples (or batch size), y_i is the prediction output and $\hat{y}_i$ is the correct output.

The rule of thumb is that if we have a small loss, the model works good.

Optimization algorithms

The main aim in the learning process is to find the best weights **minimizing** the loss function. In Deep learning fields there is an important parameter which evaluates the performance of the network: the **accuracy**. In particular, the **accuracy** measures how many times our model made the correct prediction. Indeed, if the error is close to 0, the accuracy of the model will be high. The robustness of the model increases if there is a decrease of the value of the cost function which is a non negative function.

[2]Moreover, the **mean square error** has a common implementation in the regression problem, while the **cross-entropy function** is for the classification problem. In the next chapter another loss function will be introduced: the **margin loss function** which is fundamental for CapsNet.

The optimization algorithm is a strategy that allows to minimize the error tuning the weights of the model. The idea is to initialize the network with random weights, and adjust them up and down, following the slope of the curve, in a way that the goal is fulfilled, as it is possible to see in Figure 2.3.

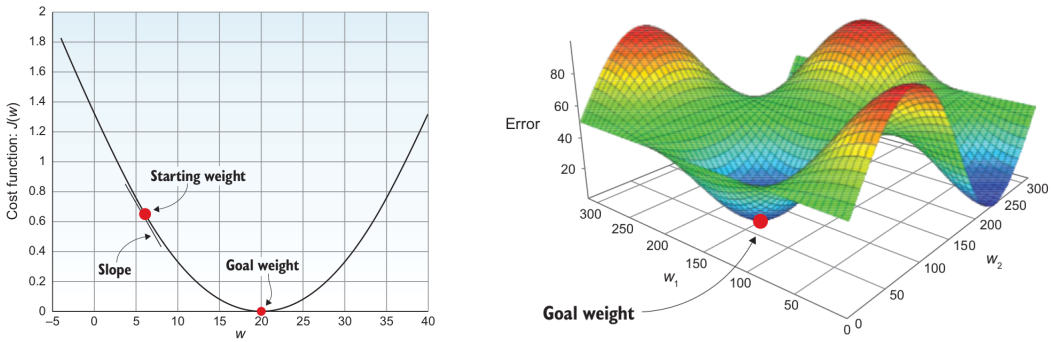


Figure 2.3: Plots of the error function versus weights in 2D and 3D planes [12]

How can we find weights? How can we adjust weights values? It is possible to solve these two answers through an iterative process, called **optimization algorithm**.

Gradient Descent (GD):

The **GD** is the simplest optimization algorithm in the literature. It is based on two concepts:

- the **gradient**: it tells us the slope of the curve (the step direction)
- the **learning rate (lr)**: it tells us the speed of the convergence (the step size)

$$\delta w_i = -\lambda_r \frac{dE}{dw_i} \tag{2.4}$$

The GD optimizer updates the weights tuning the slope of the curve in a way that the minimum is reached: it is a descent to the point with the minimum error. This operation is done in a iterative way using a step size, that is called **learning rate (lr)** : it represents how fast or slow the optimizer descends the error curve. A graphical representation of the GD algorithm is shown in Figure 2.4.

It is crucial to highlight that the learning rate is the most important hyperparameter in the design process.[3]

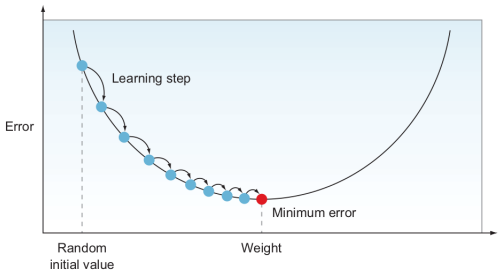


Figure 2.4: GD strategy: the minimum is reached tuning, step by step, the slope of the loss function. [12]

The GD has an important drawback: it uses the **entire** training dataset in one batch (in one step) causing a big memory consumption. This is the reason why GD is used only for small datasets.

In literature, there are other optimization algorithms that improve the properties of GD: Stochastic gradient descent(SGD) and Adam.

Stochastic gradient descent (SGD):

This algorithm divides the entire dataset into small datasets depending on the **batch-size** in a stochastic way. SGD starts its gradient descent not using only one weight starting point, but more than one in a random way allowing us to obtain several **local minimas**. After a comparison of the local minimas, the SGD get the **global minima**. In this way SGD performs better than GD solving the its drawbacks. The Figure 2.5 shows the differences between BGD and SGD strategies.

Adam

In 2017, Kingma et al. introduced Adam [13]: a novel algorithm based on **adaptive moment estimation** which outperforms other optimizers. [4]

[3]The learning rate can have several values from 0.1 to 0.0000001 and so on. If the lr is very small the training is very slow, if the lr is very high, the training is faster, but it is possible that the minimum is not reached. Our goal is to find the optimal value of lr that makes the convergence, as fast as possible.

[4]In literature there are other optimization algorithms such as Nesterov accelerate gradient, RMSprop, Adagrad and so on.

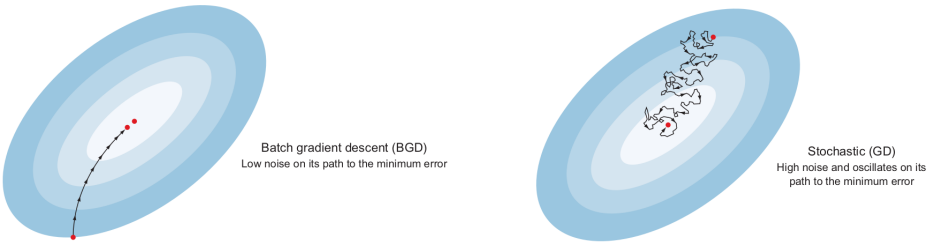


Figure 2.5: BGD and SGD [12]

3. Backpropagation error

The third step in the learning process is the backpropagation error: th feedforward process is repeated many times, updating the weights, improving the predictions in a way that prediction error is minimized (close to zero).

$$w_{i+1} = w_i + \delta w_i \tag{2.5}$$

2.3 Convolutional Neural Network

Convolutional neural network (CNN) is an advanced architecture used in image classification field. It is different from the MLPs for some aspects: the MLP learns pixel by pixel, while the CNN bases its learning process to the **feature extraction**.

The common CNN architecture is composed by an input layer, the **convolutional layers**, the FC layers and the output layer for the prediction, as shown in Figure 2.6.

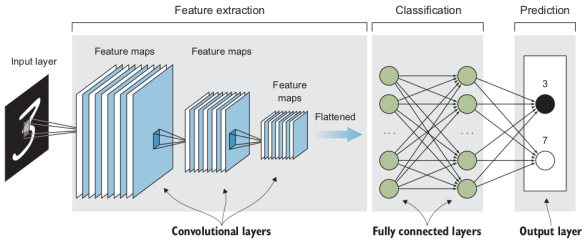


Figure 2.6: CNN architecture: the image is fed into Convolutional layers that extract the fundamental features in order to do a classification [12]

2.3.1 How does CNNs work?

The main idea of the CNNs model is to scan the input volume(image)[5], not using all pixels (as MLPs did), but through the application of the **local patterns**. In this way it is possible to store each possible small features, the feature maps. Then, in the following layers they are re-combined obtaining big features. Finally, there is an evaluation: if the object contains all right features, then it is assigned to a class with a certain probability. In Figure 2.7 there is a graphical representation that allows to understand better how CNN works.

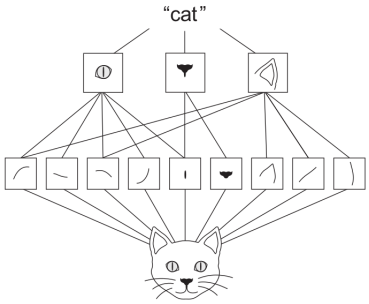


Figure 2.7: CNN strategy: in this example there is a cat at the input layer. The image is scanned into several small patterns and re-combined obtaining objects such as nose, eye and ear. Then, it is possible to classify as a "cat" a figure which has these big objects. [14]

Going in depth, the main idea is described, but in practice, how can we apply these concepts?
CNNs strategy is based on two processes: i)the convolution operation, ii)the pooling operation.

The convolution operation

At the beginning, the input volume is fed into Convolutional layers where CNNs strategy is applied. The convolution operation allows us to extracts patches[6] from its input feature map and applies the same transformation to all of these

[5]The input volume is a 3D tensor composed by two spatial axes (height,width) and a depth axis (also called channels axis). The input image can have two different axes : if axes is equal to 1 , the image has only one channel and it is in a gray-scale configuration; if axes is equal to 3 the image has three channels (RGB) and it is in a colour configuration.

[6]A patch is a window which has a size (height x width), for example it will be a matrix 3x3, 5x5, 9x9 and so on.

patches, producing an **output feature map**, as shown in Figure 2.8. [7] From a mathematical point of view, this action corresponds to a **element-wise product** between the convolution matrices.

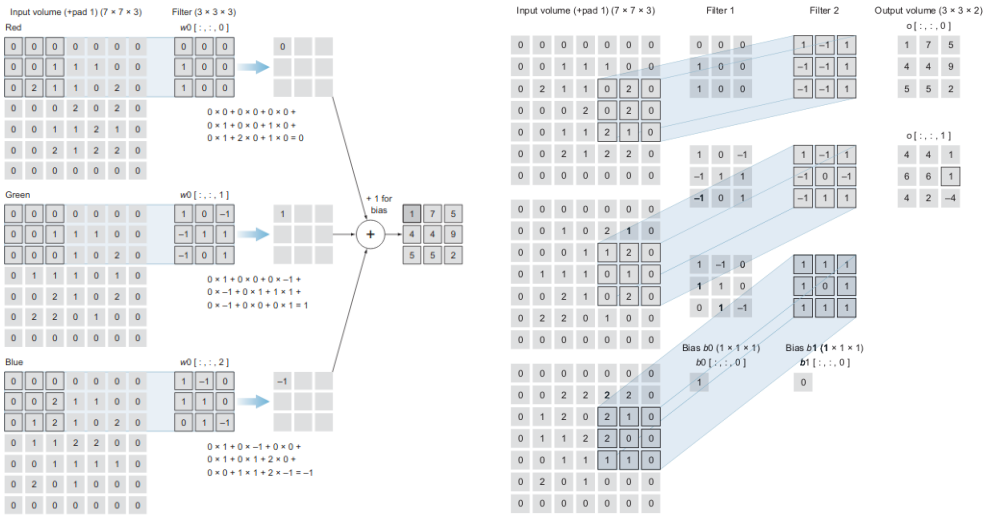


Figure 2.8: Convolution operation: it works by sliding these windows over the 3D input feature map, stopping at every possible location, and extracting the 3D patch of surrounding features. Each 3D patch is then transformed into 1D vector of shape. All of these vectors are then spatially reassembled into a 3D output map of shape. [12]

In particular, the operation of sliding is done combining two translations: from the left to right, and from the bottom to down, according to the **stride** number.[8]

In the convolution process the **border effects** (padding) of the input feature map should be considered. The padding operation is a technical way which allows that the input volume and the output feature map have the same spatial dimension adding rows and columns to the input volume around the border of an image.[9]

[7] An output feature map is a map which has a depth, that represents the number of filters, for example it will be 16, 32, 64, 256.

[8] In the sliding operation done by the convolution process, the stride represents the step to do in matrix translation. For example, if the stride = 1, it means that the sliding operation goes one pixel at time. If the stride is > 1, it goes following the same procedure, but with more pixels.

[9] An example of padding operation is the **zero-padding** where the border of the input volume is done adding zeros.

The pooling operation

After the construction of the output feature map, the pooling operation is done. It is used to down-sample feature maps, in order to reduce the number of parameters, as shown in Figure 2.9. The pooling layer is inserted between two convolutional layers. Moreover, in literature two common pooling strategies are known: Max Pooling (Figure 2.10) and Average Pooling (Figure 2.11).

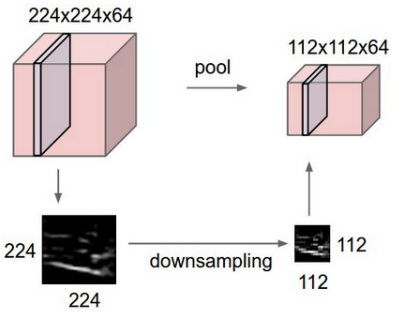


Figure 2.9: The pooling operation: it down-samples the feature maps in order to obtain a reduction to the parameters in CNNs model. [15]

- **Max Pooling:**

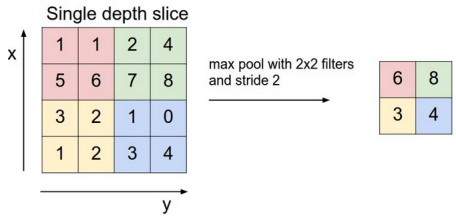


Figure 2.10: The Max pooling operation: it computes the maximum value between elements of a small matrix inside the output feature map. [15]

- **Average Pooling:**

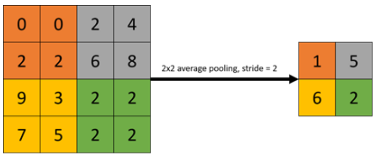


Figure 2.11: The Average pooling operation: it computes the average value between elements of a small matrix inside the output feature map. [16]

In the last few years, researchers have gone in depth with the CNNs models. G.Hinton [17] [18] was the first that has found some drawbacks and limitations in these models. For this reason, he designed a new architecture, the CapsNet [19] that will be analyzed in the next chapter.

2.4 Residual Neural Network

The DNNs are difficult to train. In 2015 K.He [20] proposed a new architecture called **ResNet** (Residual Neural Network). He changed the point of view of the classical DL models, reformulating the layers with the introduction of the residual learning framework, as shown in Figure 2.12. In this way he won the first prime in ILSVRC 2015 conference outperforming the classical DNNs optimizing better the training process.

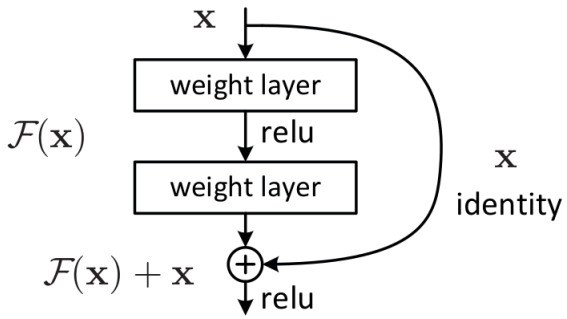


Figure 2.12: Residual learning framework [20]

The main idea of the ResNet is the introduction of the **shortcut connections**: it allows to jump over some layers. In this way the ResNet can avoid the problem of **vanishing gradients** and mitigate the **degradation problem**.[10]

2.5 Adversarial attacks

The DNNs are common in some fields such as speech recognition and image classification. Some researchers asked themselves these questions:

- *"Are DNNs not vulnerable?*

[10]The degradation problem in a particular behaviour of the DNNs that happens when there is an accuracy saturation after some epochs.

- *"Are DNNs robust architecture?"*

In 2014 Szegedy et al. [21] were the first to analyze the problem of security and robustness of the NNs introducing a new concept :**the adversarial example x'** which is an imperceptible perturbation.
If we apply an adversarial example x' to a test image, it is possible to change the network's prediction that leads to a misclassification.

The adversarial example x' has to satisfy two properties:

- $\mathcal{D}$(x, x') is small for some distance metric $\mathcal{D}$
- c(x') $\neq$ c*(x)

where $\mathcal{D}$(x, x') is the distance between the points belonged to the clean image x and the points lied at the adversarial example x'; c(x') is the new label due to adversarial example x', while c*(x) is the true label in a clean situation.
It is straightforward to declare that x' should be close to x, but x' is classified incorrectly .

How to construct an adversarial example x'?

After declaring the main concept, it is important to understand how it is possible to create an adversarial attack. In practise, crafting an adversarial example means solving a **box-constrained optimization problem** [22]. The main idea is to find a perturbation δ such that c(x+δ) $\neq$ c*(x) :

- minimizing $\| \delta \|$ subject to $\|$ x'- x $\|_p \leq \epsilon$
- maximizing the classification loss on f(x+δ)

where p is a number (from 1 to ∞) of the p-norm [11] (Table 2.2); ϵ is a sufficiently small number in a way that the p-norm is imperceptible; f(x+δ) is the neural network in an adversarial condition.

The p-norm can be written using this formula:

$$\|x\|_p = \sqrt[p]{\sum_i^n x_i^p} \tag{2.6}$$

In this way Szegedy et al. [21] demonstrated that DNNs are vulnerable to adversarial attacks

[11] If p is equal to 1, we have l_1 norm: it is a norm on the one-dimensional vector spaces and it is equal to the absolute value. If p is equal to 2, we have l_2 norm: it corresponds to the Euclidean norm and measures the Euclidean distance. If p is equal to ∞, the norm is called l_∞ which measures the maximum value in the vector space.

l_1 norm	$\|x\|_1 = \sum_i^n \|x_i\|$
l_2 norm	$\|x\|_2 = \sqrt{\sum_i^n x_i^2}$
l_∞ norm	$\|x\|_\infty = \max_{0 \leq i \leq \infty} \|x_i\|$

Table 2.2: Examples of p norms: the value of p changes from 1 to ∞ (p=1,2,∞).

Taxonomy of adversarial attacks

There are several adversarial attacks in the literature. It is possible to split them into different categories as shown in the Table 2.3.

White Box:	Full access to DNN classifier (architecture and weights)
Black Box:	Attack is crafted without access to the model
Target Attack:	Misguide the DNN to a specific class
Untarget Attack:	Don't assign a specific class to the DNN output

Table 2.3: Examples of adversarial attacks: the adversary can have access to main information of the model (white box) or not (black box); the adversary try to attack a specific class (Target attack) or not (Untarget attack).

In the third part of the thesis (from the Chap.6 to the end) we will see in depth what are the best adversarial attacks in literature, focusing also on the novel algorithms proposed until now. In addition, we will analyze a novel attack, called **Pattern attack**.

2.6 Design implementation

In the design process of the NNs (or similar architectures) the deep learning engineer has to choose and tune some **hyperparameters** in a way that it is possible to **generalize the problem**.
It is straightforward to face to three situations:

- **Underfitting**: it happens when the model is too simple to fit the data.
- **Overfitting**: it happens when the model is too able to fit the data. The NN becomes so smart memorizing every training data, but without learning the main features. The consequence of this action is the reduction of generalization and the inability to recognize other data examples .The network performs very well in training data, but it fails in testing data.

- **Generalization**: it is the situation that we want to achieve. The NN learns using training data, and tests its "experience" when new data occurs. If the model recognizes the new data in a similar way as before, with an high probability , it means that the generalization will be done.

For example, if we have a good accuracy in training data, but it is bad when we are recognizing validation data, we are in the case of **overfitting**. Luckly, we have several ways to prevent **overfitting**, such as weight regularization, dropout and data augmentation. In Figure 2.13 it is possible to see the difference for each situation.

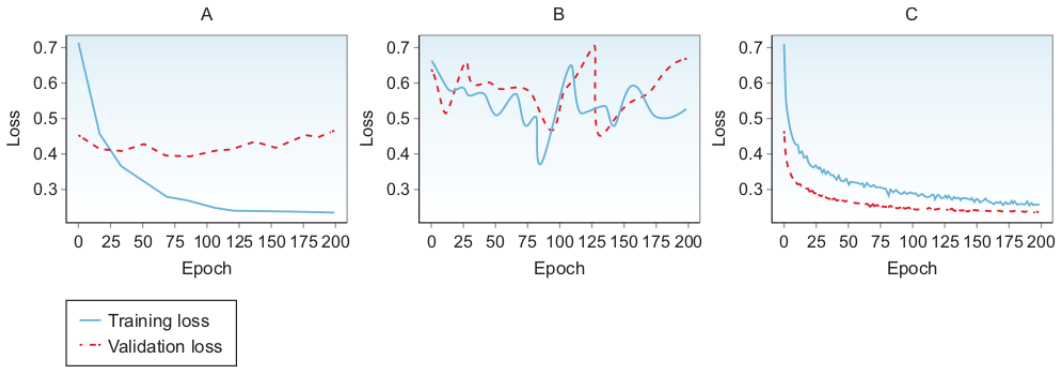


Figure 2.13: Loss / Epoch A: Overfitting; B: Underfitting; C: Generalization [12]

2.7 Remarks

This is the background chapter where there were introduced some basic concepts about the DL field such as NNs, CNNs and ResNet models. If the reader wants to going in depth to these important subjects he can consult some other references [23], [24].

Part II

CapsNet simulations

Chapter 3

State of the art : Capsule Networks

In the previous chapter we focus our attention to the CNN models, that are widespread in DL field. Despite their importance, G.Hinton [17] [18] found that CNN has some limitations and drawbacks: for this motivation he designed a new architecture, the CapsNet which outperforms the CNN architecture. In 2017 he designed the first CapsNet [19] that learns through a specific algorithm called "Dynamic Routing". After one year, Hinton proposed another view of "capsule concept" introducing the CapsNet that learns through the "EM algorithm" [25]. After the innovations proposed by Hinton, other researchers have tried to design several CapsNet models. In this Chapter I proposed the latest CapsNet models in the literature until now, but in my opinion there are others which are also valid.

3.1 What's wrong with CNN?

For many years after the LeNet-5 creation, all researches proposed new CNN architectures in order to achieve state-of-the-art performance on different datasets. Hinton [17] [18] found two crucial drawbacks in CNNs:

- Inability to understand spatial hierarchical relationship between features (simple and complex objects)
- Invariance by pooling

Picasso's problem

It is possible to understand the first drawback through an example, called **Picasso problem**, as shown in Figure 3.1. We consider an human face: it is a figure that

has face oval, two eyes, a nose and a mouth.

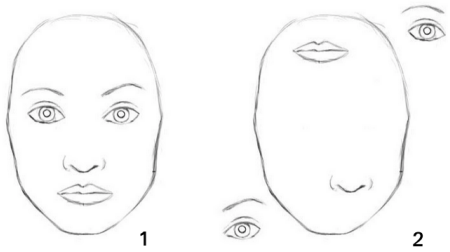


Figure 3.1: Picasso's problem: for CNNs "1" and "2" are both faces. Spatial hierarchical relationship between objects (face, nose, eyes, mouth) is not important. [26]

For CNNs the objects "1" and "2" are both faces : a presence of objects (face, eyes, nose, mouth) is sufficient to conclude that it is an human face. Orientational and relative spatial relationships between objects are not important. This conclusion is a big problem because for the human vision, that is the vision which the NN try to reproduce, the objects "1" and "2" are different, and only one is a face ("1"), but the other is only a representation of some objects (face,nose,eyes and mouth) arranged randomly in the plane which it is not possible to see in the real world, because it doesn't exist.

Pooling operation

The second drawback of the CNNs is attributable to the **pooling operation**. The Figure 3.2 shows an example of the pooling mechanism in CNN model. In the previous chapter we introduce this concept: the pooling layer, inserted between two successive Convolutional layers, is used to down-sample feature maps, in order to reduce the number of parameters. What is the cost due to a reduction in parameters?

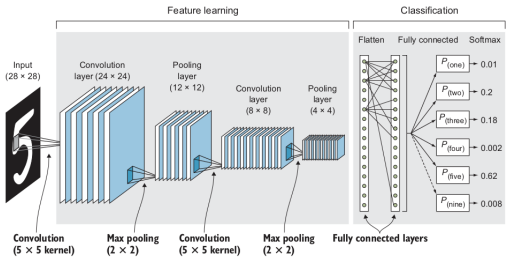


Figure 3.2: Pooling operation: spatial information is loosed after the pooling layer. [12]

For Hinton the operation of pooling is not a good strategy: a spatial information is loosed after the pooling layer.

3.2 What's a capsule?

Hinton created the **capsule** [18], in order to overcome the CNNs drawbacks. The idea came up by going to see how a computer works, in particular what are the actions that allows an image representation.

Computer Graphics vs Inverse Graphics

A computer constructs an image using the **computer graphics**: in computer's memory are stored some internal hierarchical representation of geometric data as arrays of geometrical objects and as matrices. In this way it is possible to take into account the relative positions and orientation of these objects. A process, called **rendering** converts this representation into an image.
The idea of Hinton [26] is to do the opposite operation of the computer graphics: **inverse graphics** as human brain does (Figure 3.3). In fact, when we see an image our brain splits every objects of the image in a way that, when we see the same image, but with a different angle, we immediately recognize it.

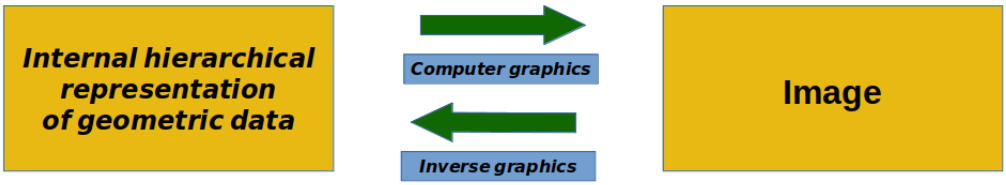


Figure 3.3: Computer graphics and inverse graphics

Following the **inverse graphics** process, Hinton designed a **capsule** where storing the internal hierarchical representation of geometric data preserving hierarchical pose relationships between object parts and equivariance.

Figure 3.4: Capsule's configuration: at the left of the figure there is a pose vector while a pose matrix is represented at the right. [27]

A **capsule** is, for definition, a group of neurons where encapsulates properties of single entity [27]. It can be represented through two configurations: i)pose vector, ii) pose matrix [25], as shown in Figure 3.4.
Going into details, we consider a first configuration of the **pose** [1]in order to understand better how the capsule works. The Figure 3.5 is a graphical representation of the Capsule: if we have a capsule as in a vector configuration, each capsule has a **vector output** such that [28]:

- **Length**: it estimates the probability of presence
- **Orientation**: it estimates the object's pose parameters (instantiation parameters)

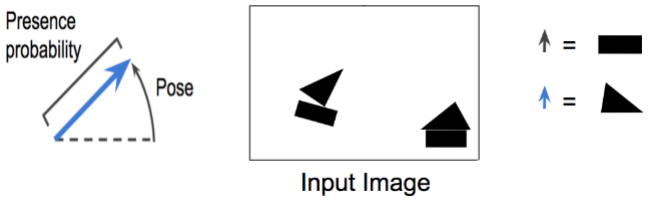


Figure 3.5: Capsule representation: each capsule has a vector output in order to estimate the object's pose parameters and the probability of presence. [28]

What was the genesis of the "capsule" ?

In principle, Hinton has found this idea many year before, but there were some problems to realize it: there was no algorithm to implement the CapsNet in a successful way and it was difficult to make it in hardware design. Luckly, thanks to the invention of **GPU** [2] the task was solved in a good way.

3.3 Shallow Caps

In 2017, G.Hinton et al. [19] proposed a new kind of architecture that achieves state-of-the-art performance on MNIST [29] through an **iterative routing-by-agreement** mechanism.

[1]A pose gives us a representation of 3D objects taking into account the relations between objects using translation and rotation.

[2]GPU is a new device able to accelerate the creation of images and to make some computation graphics tasks.

3.3.1 How does a capsule work?

How can the capsule learn? What is the learning process?

The learning process of the capsule is composed by several operations:

- **Affine transform:**

$$\hat{u}_{j|i} = W_{ij} u_i \tag{3.1}$$

  The "prediction vectors" $\hat{u}_{j|i}$ is performed by the multiplication of the "output" u_i of the capsule i by a weight matrix W_{ij}.

- **Weighted sum:**

$$s_j = \sum_i c_{ij} \hat{u}_{j|i} \tag{3.2}$$

  The capsule s_j is done computing the weighted sum between the "coupling coefficients" c_{ij} and the "prediction vectors" $\hat{u}_{j|i}$.

- **Non linear activation: squash function**

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|} \tag{3.3}$$

  The squashing operation is applied to the capsule s_j. The "vector output" v_j of the capsule j is obtained through the application of the **squash function** shrunking the **length** of the vectors: the short vectors go to 0, while the long vectors go to 1.

A capsule is different from the traditional neuron: in the first configuration the input/output of the capsule is a vector, while the input/output of the neuron is a scalar.
According for what is introduced to the previous chapter, in the traditional neuron configuration the **weighted sum** is obtained through $z = \sum x_i \cdot w_i + b$ while the non linear activation (sigmoid, tanh, ReLu and so on) is done by $\hat{y} = a = f(z)$.

3.3.2 Dynamic routing algorithm

Putting these concepts together Hinton proposed the **dynamic routing algorithm** which involves capsules according to an **iterative agreement**: it allows capsules to communicate with each other in order to create images in a good way. The Figure 3.6 shows, step by step, how to apply the dynamic routing algorithm.

Routing algorithm

procedure ROUTING($\hat{\boldsymbol{u}}_{j|i}, r, l$)
  for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
  for r iterations **do**
    for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ `softmax`
    for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
    for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ `squash`
    for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
  return $\mathbf{v}_j$

Figure 3.6: Dynamic routing algorithm: it is a route agreement process between capsules [19]

3.3.3 The architecture

The **Shallow Caps** is composed into two sections: i)the Encoder part, ii)the Decoder part. The Figure 3.7 shows an example of CapsNet architecture applied to MNIST dataset. [29]

i) The Encoder part:

The Encoder part is the core of the CapsNet where the "dynamic routing algorithm" is performed causing an image classification. It is splitted into several layers:

- Input Layer: is the input volume (image)
- Conv2D : is the first convolutional layer which extracts the feature maps
- Primary Caps: they transform the feature maps into capsules
- Digit Caps: where the "dynamic routing algorithm" is performed causing an image classification

The loss function used in the "encoder part" is the **margin loss**:

$$L_k = T_k \max(0, m^+ - \|v_k\|)^2 + \lambda(1 - T_k)\max(0, \|v_k\| - m^-)^2 \tag{3.4}$$

The margin loss L_k is the sum of the all losses concerning all capsules. The parameters are: $T_k = 1$; $m^+ = 0.9$; $m^- = 0.1$; $\lambda = 0.5$.

ii) The Decoder part:

Instead, the Decoder part is fundamental for the reconstruction of the image in order to compare the initial image to the image, reconstructed thanks to the

classification. In the Shallow Caps configuration, it is composed by several FC layers connected each other. The loss function used in the "decoder part" is the $\mathbf{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$.

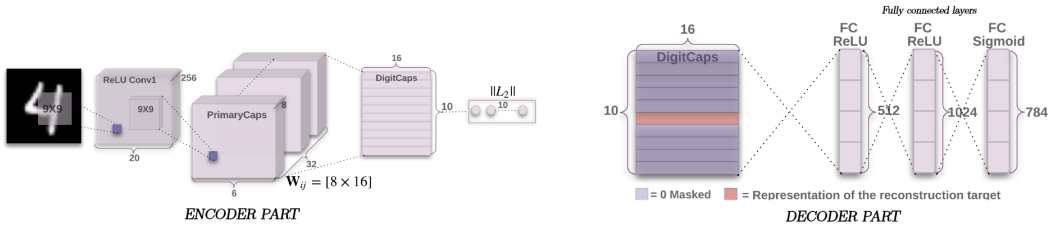


Figure 3.7: CapsNet architecture[19]: encoder and decoder part on MNIST dataset.

3.3.4 ShallowCaps Drawbacks

The Shallow Caps has several drawbacks[30] to take into account:

- DR is computationally expensive procedure
- Higher costs of training and inference time
- Poor learning in the middle layers

3.4 Deep Caps

J. Rajasegaran et al. [30] studied in depth the properties of CNNs models finding a way to overcome the ShallowCaps drawbacks. In 2019, they proposed a novel architecture called **DeepCaps** which outperform the State-of-the-art of the ShallowCaps [19] results on CIFAR10 [31], SVHN [32] and Fashion MNIST [33]. DeepCaps allow us to reduce:

- the number of routing iterations in the initial layers
- the number of parameters involved

3.4.1 3D dynamic routing algorithm

In the DeepCaps the dynamic routing is performed using the 3D convolution in the middle layers. This particular choice guarantees an improvement of the performance of the CapsNet for complex image datasets. The Figure 3.8 shows, step by step, how to apply the 3D dynamic routing algorithm.

Dynamic Routing using 3D convolution

procedure ROUTING
Require: $\mathbf{\Phi}^l \in \mathbb{R}^{(w^l, w^l, c^l, n^l)}$, r and c^{l+1}, n^{l+1}
  $\tilde{\mathbf{\Phi}}^l \leftarrow \texttt{Reshape}(\Phi_l) \in \mathbb{R}^{(w^l, w^l, c^l \times n^l, 1)}$
  $\mathbf{V} \leftarrow \texttt{Conv3D}(\tilde{\mathbf{\Phi}}^l) \in \mathbb{R}^{(w^{l+1}, w^{l+1}, c^l, c^{l+1} \times n^{l+1})}$
  $\tilde{\mathbf{V}} \leftarrow \texttt{Reshape}(\mathbf{V}) \in \mathbb{R}^{(w^{l+1}, w^{l+1}, n^{l+1}, c^{l+1}, c^l)}$
  $\mathbf{B} \leftarrow \mathbf{0} \in \mathbb{R}^{(w^{l+1}, w^{l+1}, c^{l+1}, c^l)}$
  Let $p \in w^{l+1}, q \in w^{l+1}, r \in c^{l+1}$ and $s \in c^l$
  for i iterations **do**
    for all p, q, r, $k_{pqrs} \leftarrow \texttt{softmax\_3D}(b_{pqrs})$
    for all s, $S_{pqr} \leftarrow \sum_s k_{pqrs} \cdot \tilde{V}_{pqrs}$
    for all s, $\hat{S}_{pqr} \leftarrow \texttt{squash\_3D}(S_{pqr})$
    for all s, $b_{pqrs} \leftarrow b_{pqrs} + \hat{S}_{pqr} \cdot \tilde{V}_{pqrs}$
  return $\mathbf{\Phi}^{l+1} = \hat{\mathbf{S}}$

Figure 3.8: 3D dynamic routing algorithm [30]

3.4.2 The architecture

According to the ShallowCaps, the DeepCaps is composed by two parts: i) the Encoder part and ii) the Decoder part. The Figure 3.9 shows an example of DeepCaps architecture.

i) The Encoder part

Rajasegaran introduced the concept of **CapsCell**: it is a structure which contains several Conv2D layers. Indeed, the Encoder part is based on 4 "CapsCells" where each "CapsCell" has 4 Conv2D layers. In the fourth "CapsCell" one Conv2D layer is replaced by the **ConvCapsuleLayer3D** which is the core of DeepCaps where the **3D dynamic routing** is performed. At the end of the Encoder part there is a DigitCaps layer.

ii) The Decoder part

Rajasegaran proposed an innovative decoder based on **deconvolutional** layers that make better the reconstruction process compared to the FC ones (used in ShallowCaps), thanks to a more spatial relationships. Furthermore, in DeepCaps architecture only the activity vectors of the predicted class is fed into the Decoder.

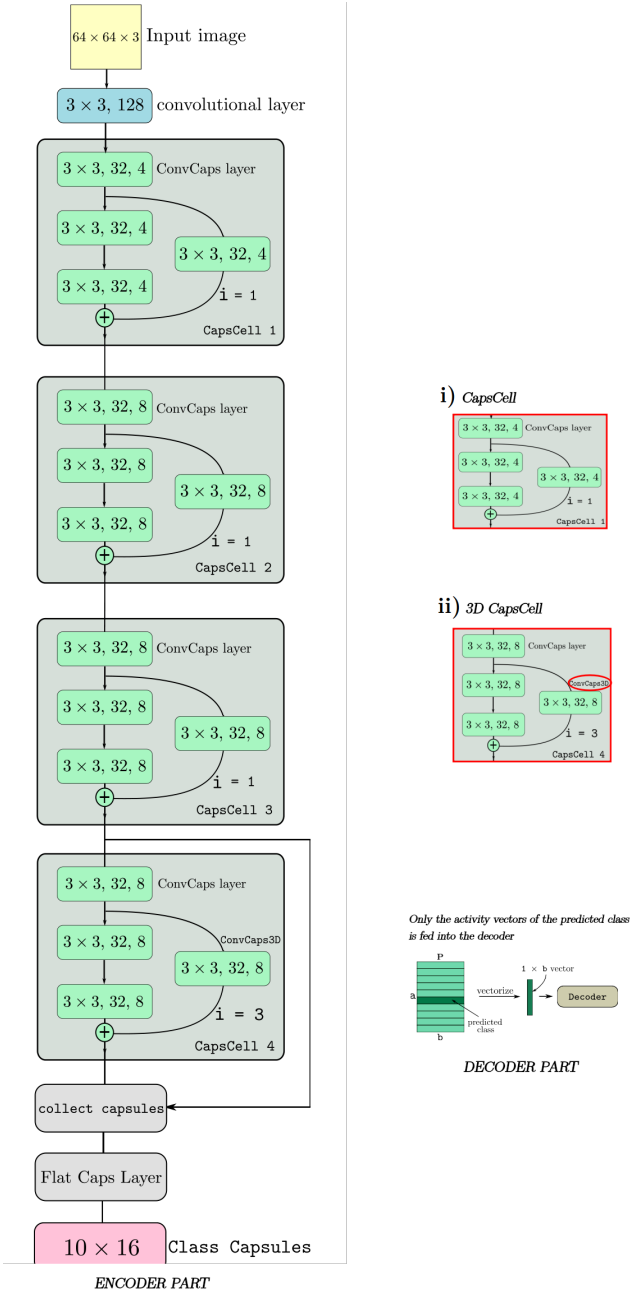


Figure 3.9: DeepCaps architecture[30]: i) and ii) are the basic components of the Encoder part. At the bottom of the figure it is possible to see how the decoder works.

3.5 Yao Caps

At the beginning of 2020, Yao Tsai et al. [34] introduced a novel routing algorithm: it is based on an **Inverted Dot-product Attention Routing**, where the agreement is performed through dot products. The **pose** of capsule is in a **matrix form**, but it is different from the matrix pose proposed by Hinton [25]. In Hinton case, the activation probability is determined by the EM algorithm, while in Yao Caps it is not represented. Moreover, the **loss function** is no longer the margin loss, as it happens in ShallowCaps and DeepCaps, but **Cross Entropy** or **Binary Cross Entropy** are used. Figure 3.10 shows, step by step, how to apply the Inverted Dot-product Attention Routing algorithm.

Inverted Dot-product Attention Routing algorithm returns updated **poses** of the capsules in layer $L+1$ given **poses** in layer L and $L+1$ and **weights** between layer L and $L+1$.

1: **procedure** INVERTED DOT-PRODUCT ATTENTION ROUTING($\mathbf{P}^L$, $\mathbf{P}^{L+1}$, $\mathbf{W}^L$)
2: for all capsule i in layer L and capsule j in layer $(L+1)$: $\mathbf{v}_{ij}^L \leftarrow \mathbf{W}_{ij}^L \cdot \mathbf{p}_i^L$ ▷ vote
3: for all capsule i in layer L and capsule j in layer $(L+1)$: $a_{ij}^L \leftarrow {\mathbf{p}_j^{L+1}}^\top \cdot \mathbf{v}_{ij}^L$ ▷ agreement
4: for all capsule i in layer L: $r_{ij}^L \leftarrow \exp(a_{ij}^L) \,/\, \sum_{j'} \exp(a_{ij'}^L)$ ▷ routing coefficient
5: for all capsule j in layer $(L+1)$: $\mathbf{p}_j^{L+1} \leftarrow \sum_i r_{ij}^L \mathbf{v}_{ij}^L$ ▷ pose update
6: for all capsule j in layer $(L+1)$: $\mathbf{p}_j^{L+1} \leftarrow \texttt{LayerNorm}(\mathbf{p}_j^{L+1})$ ▷ normalization
7: **return** $\mathbf{P}^{L+1}$

Figure 3.10: Inverted Dot-product Attention Routing algorithm [34]

After the routing process, a **Layer Normalization** [35] (2016) is added in order to improve the convergence. This is another difference with ShallowCaps and DeepCaps, because in these cases the Layer Normalization is not present.

3.5.1 The architecture

YaoCaps architecture is a combination of **ResNet blocks** and **Capsule layers**. The input image is fed into **ResNet blocks** to produce the feature maps, after that it is transformed into capsules by **PrimaryCaps** layer, and then it goes into **Convolutional Capsules** where an **Inverted Dot-product Attention Routing algorithm** is done.

The Figure 3.11 shows an example of YaoCaps architecture: there is only the Encoder part.

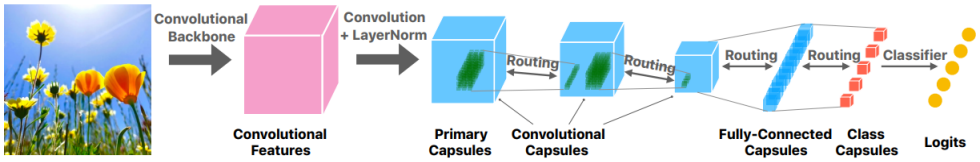


Figure 3.11: YaoCaps architecture [34]

3.6 FM Caps

At the middle of 2020, L.Zhao et al. [36] proposed a novel algorithm, which is based on **pairwise agreement mechanism**. The idea came out thanks to the previous work carried out by Rendle et al. [37], dealing with the feature interactions of "factorization machines". Figure 3.12 shows, step by step, how to apply the FM algorithm.

FM Agreement

Input: prediction vectors $\hat{\mathbf{U}}_j = \left(\hat{\mathbf{u}}_{j|1}, \ldots, \hat{\mathbf{u}}_{j|n}\right)$

Output: $\hat{\mathbf{p}}_j, \hat{a}_j$

1: $\hat{\mathbf{u}}_{j|i} \leftarrow \mathit{L2Normalize}(\hat{\mathbf{u}}_{j|i})$ $\quad \forall i$

2: $\hat{\mathbf{s}}_j \leftarrow \frac{1}{2n}\left(\sum_{i=1}^{n} \hat{\mathbf{u}}_{j|i} \odot \sum_{i=1}^{n} \hat{\mathbf{u}}_{j|i} - \sum_{i=1}^{n} \hat{\mathbf{u}}_{j|i} \odot \hat{\mathbf{u}}_{j|i}\right)$

3: $\hat{\mathbf{p}}_j \leftarrow \frac{\hat{\mathbf{s}}_j}{\|\hat{\mathbf{s}}_j\|}$

4: $\hat{a}_j \leftarrow \sum_{f=1}^{k} \hat{s}_{j,f}$

Figure 3.12: FM algorithm [36]

3.6.1 The architecture

The architecture is similar to the YaoCaps, but in this case the output of PrimaryCaps layer goes into 3 **CapsLayers** where **FM Agreement** is done. Figure 3.13 shows an example of FM architecture.

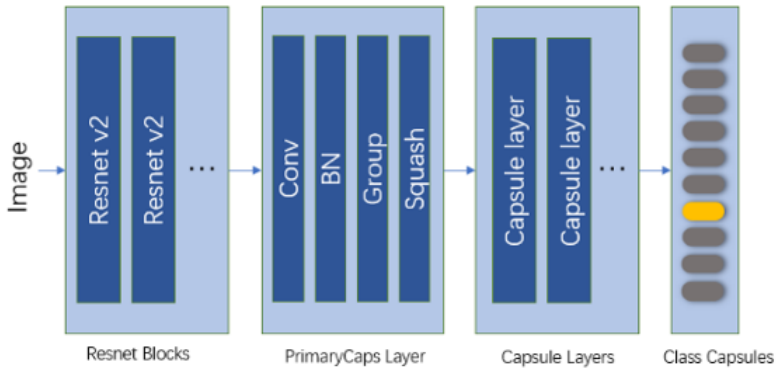


Figure 3.13: FMCaps architecture [36]

3.7 Remarks

Researchers continue to investigate ways to improve accuracy in the DL field. In the literature there are other types of CapsNet, very promising such as: **STAR-Caps** [38] ,**Trans-Caps** [39], **CapsVB** [40] and **Efficient CapsNet** [41].

Chapter 4

CapsNet analysis

After an introduction about the best CapsNet architectures (in Chap. 3), it is useful to analyze them applying to several datasets taking into account the training process and the final accuracy. At the beginning the analysis goes to the common datasets such as SVHN, CIFAR10 and CIFAR100. After that, we go next introducing three novel datasets: CINIC10 which is used in this chapter and other ones, that are more important and for this reason there is a section for each one (MLRS and COVID19).

4.1 Implementation

This is the implementation on several datasets using different CapsNet models.

We follow this procedure in order to obtain the best performances:

i. Split the datasets according to the **Pareto optimal rule** (80/20), in my case I decide to split according 85/15, where's possible

ii. Run several simulations with different architectures choosing the one that has the best result

iii. Plot loss and accuracy functions due to simulation process

iv. Count the time elapsed in a training process

v. As soon as I get the best result I decide to run all the models 3 times and I report their accuracy by "average±std"

vi. Plot the real images and the reconstructed images (after decoder) in order to take a comparison

vii. Repeat i) - vi) for all datasets and CapsNet models, where's possible

It is important to focus our attention on point iii): in the plot (accuracy vs epochs) the curve of validation Capsnet accuracy should be less than the curve of training Capsnet accuracy. This action should be fulfilled for all the time simulation, but in particular for the end region of the plot. At the initial time we can have some instability phenomena which end after some epochs (according to the model and dataset). Indeed, at the end region the model will be stabilized and the accuracy can't grown so much.
This fundamental "rule of thumb" allows us to improve the performance of our models in order to generalize better the configuration: it is important to remember it because it will be useful both in this chapter and in the next.

Going in depth, the training process is done through a **data augmentation** for all CapsNets models.
In the ShallowCaps and DeepCaps architectures we use the margin loss for the encoder part, while the MSE is used for the decoder part. In addiction, we use the Adam optimizer.
In the YaoCaps and FMCaps architectures we have only the "encoder part" : the YaoCaps uses the cross-entropy while the FMCaps uses the sparse categorical cross-entropy. Both YaoCaps and FMCaps use the SGD optimizer.
Furthermore, **Keras**[42] and **TensorFlow**[43] libraries are used for the development of ShallowCaps, DeepCaps and FMCaps. For YaoCaps the **PyTorch**[44] framework is implemented. The models were trained on NVIDIA Ge Force RTX 2080 Ti. [1]

4.2 SVHN

The SVHN[45] (Figure 4.1) is the first dataset where CapsNets are applied. It is similar to the MNIST [29] because the images represent digits, but it is more difficult computing the classification problem.

The SVNH is obtained from house numbers in Google Street View images. The images contain overlapping digits and distracting features. This dataset consists of 10 classes, the models are trained on the 73.275 training images and evaluated on the 26.032 validation images. The input image size of SVHN is 32-by-32 RGB pixels.
Indeed, we evaluate the SVHN on two CapsNets architectures: i)Shallow Caps and ii)DeepCaps.

[1] In this context, all simulations are performed on **Tu Wien Platform** which is composed by several GPUs.

Figure 4.1: SVHN Dataset [32]

4.2.1 ShallowCaps evaluation

This is the implementation of the ShallowCaps on SVHN dataset. The input layer has size by 32x32x3. The model is trained for 100 epochs using the batch size equal to 128; the number of parameters of the model is about 27.8 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by three FC layers, useful to reconstruct the images.

Model
Input layer: (input_shape=(32,32,3))
Conv2D: (filters=1024, kernel_size=9, strides=1, padding='valid', activation='relu') + BatchNormalization
PrimaryCaps: (dim_capsule=8, n_channels=32, kernel_size=9, strides=2, padding='valid')
DigitCaps: (num_capsule=n_classes=10, dim_capsule=16)
Mask: digitCaps=10*16=160
Decoder: Dense(512, activation='relu', input_dim=16*10)) Dense(1024, activation='relu')) Dense(np.prod(32,32,3), activation='sigmoid')) Reshape(target_shape=(32,32,3))

Figure 4.2: Implementation of the ShallowCaps on SVNH

According to the configuration written in Figure 4.2, the average test accuracy obtained is 94.96% $\pm$ 0.05, as shown in Figure 4.3.

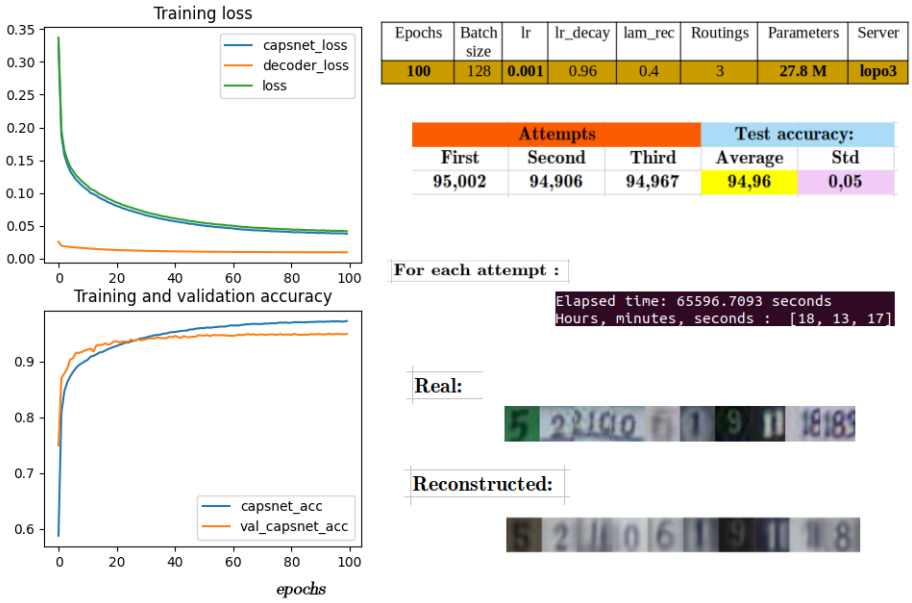


Figure 4.3: ShallowCaps on SVHN: Plots and results

4.2.2 DeepCaps evaluation

This is the implementation of the DeepCaps on SVHN dataset. The input layer has size by 32x32x3: it is resized to 64x64x3 that allows us to go down deeper. The model is trained for 100 epochs using the batch size equal to 160; the number of parameters of the model is about 13.4 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by five Deconvolution 2D layers, useful to reconstruct the images.

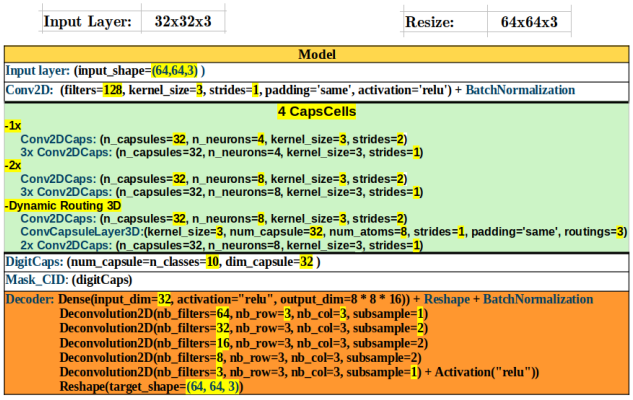
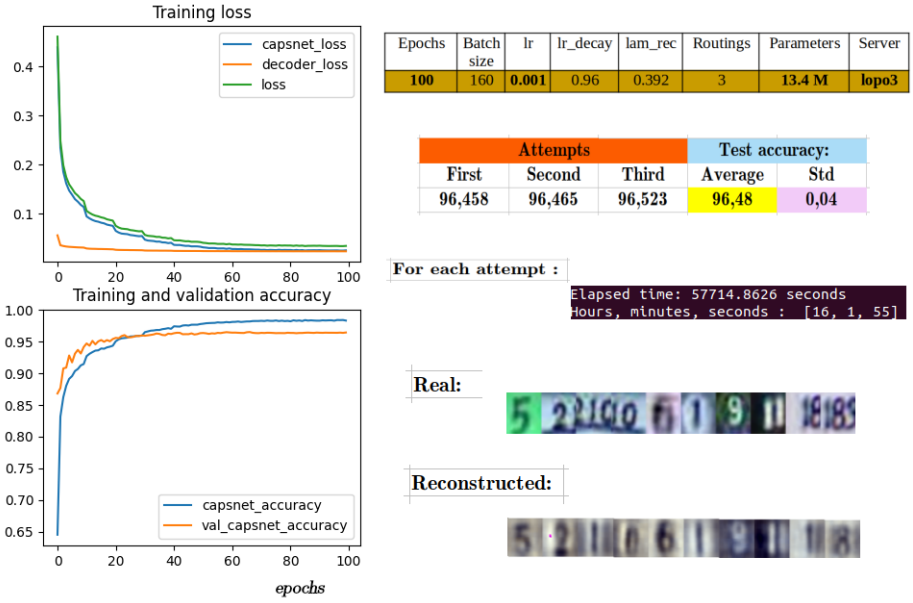


Figure 4.4: Implementation of the DeepCaps on SVNH

According to the configuration written in Figure 4.4, the average test accuracy obtained is 96.48% ± 0.04, as shown in Figure 4.5.



Epochs	Batch size	lr	lr_decay	lam_rec	Routings	Parameters	Server
100	160	**0.001**	0.96	0.392	3	**13.4 M**	**lopo3**

Attempts			Test accuracy:	
First	Second	Third	Average	Std
96,458	96,465	96,523	96,48	0,04

Figure 4.5: DeepCaps on SVHN: Plots and results

4.3 CIFAR10

The CIFAR10[46] (Figure 4.6) is the second dataset where CapsNets are applied.

This dataset consists of 10 classes, the models are trained on the 50.000 training images and evaluated on the 10.000 validation images. The input image size of CIFAR10 is 32-by-32 RGB pixels.
Indeed, we evaluate the CIFAR10 on four CapsNets architectures: i)Shallow Caps , ii)DeepCaps, iii)YaoCaps and iv)FMCaps.

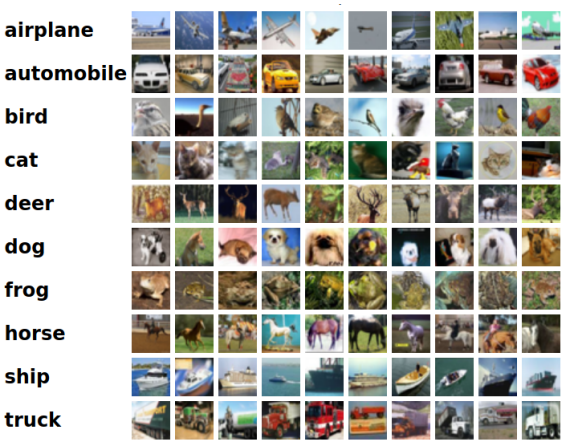


Figure 4.6: CIFAR10 Dataset [31]

4.3.1 ShallowCaps evaluation

This is the implementation of the ShallowCaps on CIFAR10 dataset. The input layer has size by 32x32x3. The model is trained for 50 epochs using the batch size equal to 100; the number of parameters of the model is about 11.7 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. The decoder is composed by three FC layers, useful to reconstruct the images.

Model
Input layer: (input_shape=(32,32,3))
Conv2D: (filters=256, kernel_size=9, strides=1, padding='valid', activation='relu')
PrimaryCaps: (dim_capsule=8, n_channels=32, kernel_size=9, strides=2, padding='valid')
DigitCaps: (num_capsule=n_classes=10, dim_capsule=16)
Mask: digitCaps=10*16=160
Decoder: Dense(512, activation='relu', input_dim=16*10)) Dense(1024, activation='relu')) Dense(np.prod(32,32,3), activation='sigmoid')) Reshape(target_shape=(32,32,3))

Figure 4.7: Implementation of the ShallowCaps on CIFAR10

According to the configuration written in Figure 4.7, the average test accuracy obtained is 70.68% $\pm$ 0.53, as shown in Figure 4.8.

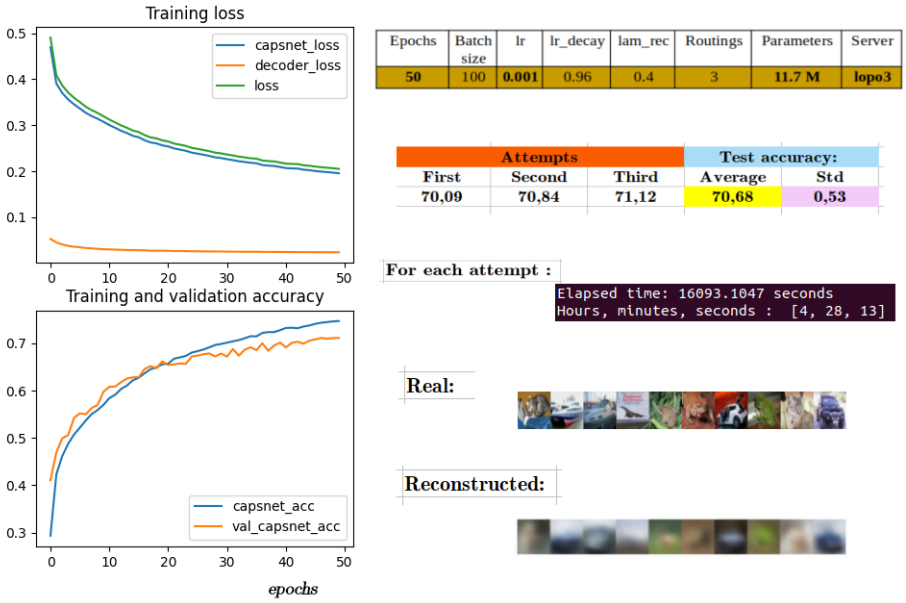


Figure 4.8: ShallowCaps on CIFAR10: Plots and results

4.3.2 DeepCaps evaluation

This is the implementation of the DeepCaps on CIFAR10 dataset. The input layer has size by 32x32x3: it is resized to 64x64x3 that allows us to go down deeper. The model is trained for 50 epochs using the batch size equal to 128; the number of parameters of the model is about 13.4 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by five Deconvolution 2D layers, useful to reconstruct the images.

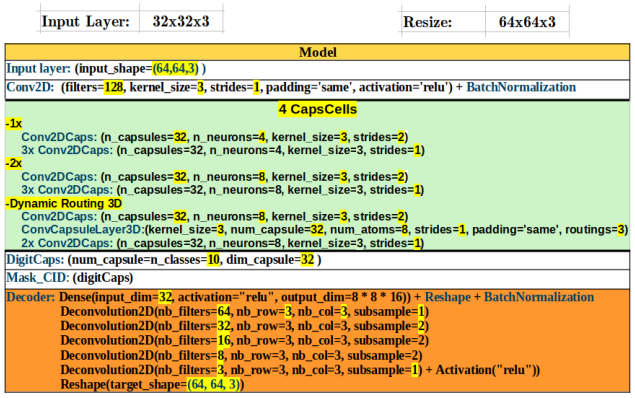


Figure 4.9: Implementation of the DeepCaps on CIFAR10

According to the configuration written in Figure 4.9, the average test accuracy obtained is 89.00% ± 0.57, as shown in Figure 4.10.

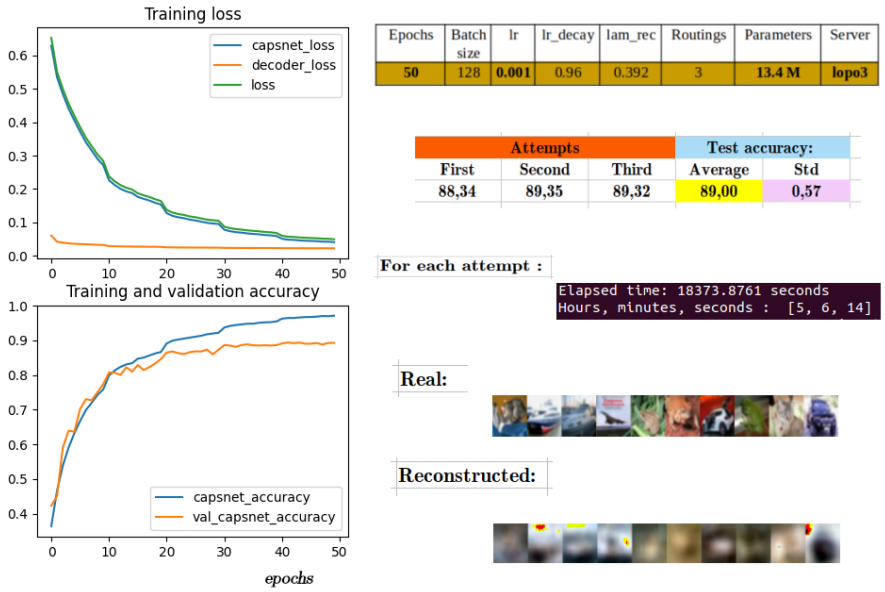


Figure 4.10: DeepCaps on CIFAR10: Plots and results

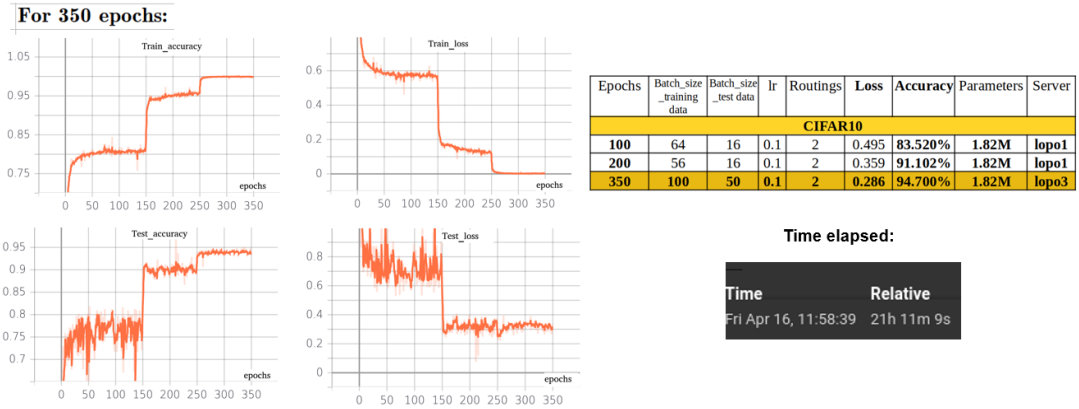
4.3.3 YaoCaps evaluation

This is the implementation of the YaoCaps on CIFAR10 dataset. The input layer has size by 32x32x3. In this case, several attempts have been performed. The best result is obtained when the model is trained for 350 epochs: the batch size for training data is equal to 100, while the batch size for testing data is 50. Moreover, the number of parameters of the model is about 1.82 million and the number of routing process is 2. The learning rate starts from 0.1 and decreases from 150 to 250 epochs. It is important to point out that in this case there is only the encoder part.

Model
Input layer: (input_shape=(32,32,3))
Backbone: ResNet blocks
PrimaryCaps: (num_caps=32, caps_dim=16, kernel_size=1, strides=1, padding='same')
Convolutional_Capsules Capsules1: (num_caps=32, caps_dim=16, kernel_size=3, strides=2) Capsules2: (num_caps=32, caps_dim=16, kernel_size=3, strides=1)
Class_Capsules: (caps_dim= 16, num_caps=n_classes=10)

Figure 4.11: Implementation of the YaoCaps on CIFAR10

According to the configuration written in Figure 4.11, the best test accuracy obtained is 94.70%, as shown in Figure 4.12. Moreover, in all plots there are two steps, when the learning rate changes.



Epochs	Batch_size _training data	Batch_size _test data	lr	Routings	Loss	Accuracy	Parameters	Server
CIFAR10								
100	64	16	0.1	2	0.495	**83.520%**	**1.82M**	**lopo1**
200	56	16	0.1	2	0.359	**91.102%**	**1.82M**	**lopo1**
350	**100**	**50**	**0.1**	**2**	**0.286**	**94.700%**	**1.82M**	**lopo3**

Figure 4.12: YaoCaps on CIFAR10: Plots and results

4.3.4 FMCaps evaluation

This is the implementation of the FMCaps on CIFAR10 dataset. The input layer has size by 32x32x3. In this case, several attempts have been performed. The best result is obtained when the model is trained for 150 epochs: the batch size is equal to 128. Moreover, the number of parameters of the model is about 0.97 million. The learning rate starts from 0.1 and decreases from 81 to 122 epochs. It is important to point out that in this case, as in YaoCaps, there is only the encoder part.

Model
Input layer: (input_shape=(32,32,3))
Backbone: 25x ResNet blocks
PrimaryCaps: (groups=4, atoms=16, kernel_size=5, strides=2, padding='same') + BatchNormalization
MultiCapsules
CapsLayer1:
CapsuleTransformDense(num_capsule=32) + BatchNormalization + Activation + LastFMPool
CapsLayer2:
CapsuleTransformDense(num_capsule=16) + BatchNormalization + Activation + LastFMPool
CapsLayer3:
CapsuleTransformDense(num_capsule=n_classes=10)+ BatchNormalization + Activation + LastFMPool

Figure 4.13: Implementation of the FMCaps on CIFAR10

According to the configuration written in Figure 4.13, the best test accuracy obtained is 93.48%, as shown in Figure 4.14. Moreover, in all plots there is a big step, when the learning rate changes.

FM Caps	Epochs	Batch size	lr	Loss	Accuracy	Parameters	Server
CIFAR10							
	50	128	0.1	0.8032	**83.26%**	**907 k**	**lopo3**
	100	128	0.1	0.50884	**92.62%**	**907 k**	**lopo3**
	150	128	0.1	0.46793	**93.48%**	**907 k**	**lopo3**

For 150 epochs:

```
Elapsed time: 4339.7053 seconds
Hours, minutes, seconds :  [1, 12, 20]
```

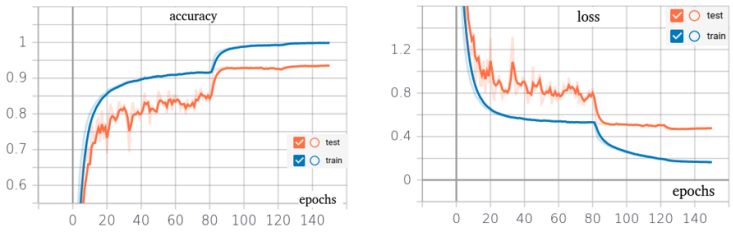


Figure 4.14: FMCaps on CIFAR10: Plots and results

4.4 CIFAR100

The CIFAR100[46] is an extended version of CIFAR10 dataset. Indeed, CIFAR100 consists of 100 classes: it has 10 times the number of classes of CIFAR10. The difficulty of CIFAR100 is that we only have 500 images per class in the training process, instead of 5000 ones per class as it happens in CIFAR10 case.

The models are trained on the 50.000 training images and evaluated on the 10.000 validation images. The input image size of CIFAR100 is 32-by-32 RGB pixels.

Indeed, we evaluate the CIFAR100 on four CapsNets architectures: i)Shallow Caps , ii)DeepCaps, iii)YaoCaps and iv)FMCaps.

4.4.1 ShallowCaps evaluation

According to the standard procedure, following the configuration proposed by Sabour et al. [19] it seems there is no way to obtain good performances. J.Gu [47] modified a standard ShallowCaps configuration, adding ResNet18 backbone before the Primary Capsules for CIFAR10 case.

Following this suggestion in a more complex dataset such as CIFAR100, the accuracy of ShallowCaps goes to 51 %.

4.4.2 DeepCaps evaluation

This is the implementation of the DeepCaps on CIFAR100 dataset. The input layer has size by 32x32x3. The model is trained for 50 epochs using the batch size equal to 8; the number of parameters of the model is about 23.3 million. The learning rate starts from 0.0001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by four Deconvolution 2D layers.

```
Model
Input layer: (input_shape=(32,32,3) )
Conv2D:  (filters=128, kernel_size=3, strides=1, padding='same', activation='relu') + BatchNormalization
4 CapsCells
-1x
    Conv2DCaps: (n_capsules=32, n_neurons=4, kernel_size=3, strides=2)
    3x Conv2DCaps: (n_capsules=32, n_neurons=4, kernel_size=3, strides=1)
-2x
    Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=2)
    3x Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=1)
-Dynamic Routing 3D
    Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=2)
    ConvCapsuleLayer3D:(kernel_size=3, num_capsule=32, num_atoms=8, strides=1, padding='same', routings=3)
    2x Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=1)
DigitCaps: (num_capsule=n_classes=100, dim_capsule=32 )
Mask_CID: (digitCaps)
Decoder: Dense(input_dim=32, activation="relu", output_dim=8 * 8 * 16)) + Reshape + BatchNormalization
         Deconvolution2D(nb_filters=64, nb_row=3, nb_col=3, subsample=1)
         Deconvolution2D(nb_filters=32, nb_row=3, nb_col=3, subsample=2)
         Deconvolution2D(nb_filters=16, nb_row=3, nb_col=3, subsample=2)
         Deconvolution2D(nb_filters=3, nb_row=3, nb_col=3, subsample=1) + Activation("relu"))
         Reshape(target_shape=(32, 32, 3))
```

Figure 4.15: Implementation of the DeepCaps on CIFAR100

According to the configuration written in Figure 4.15, the average test accuracy obtained is 59.25% ± 0.24, as shown in Figure 4.16.

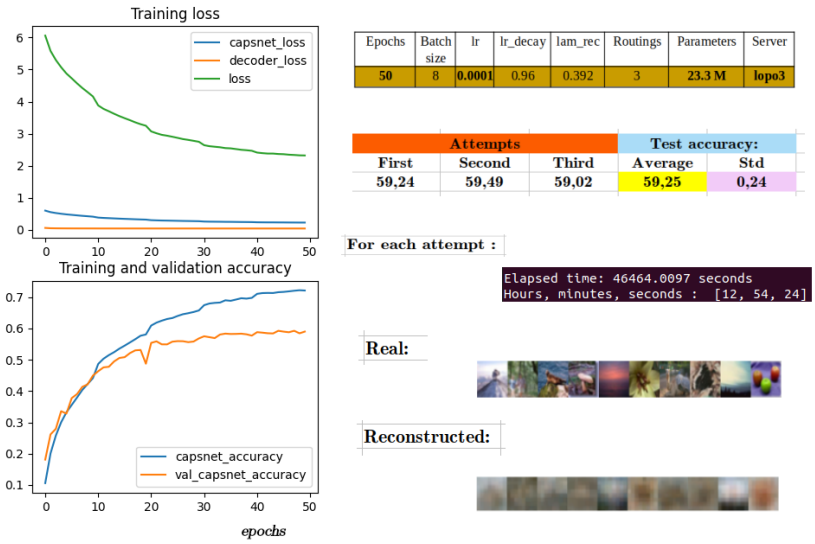


Figure 4.16: DeepCaps on CIFAR100: Plots and results

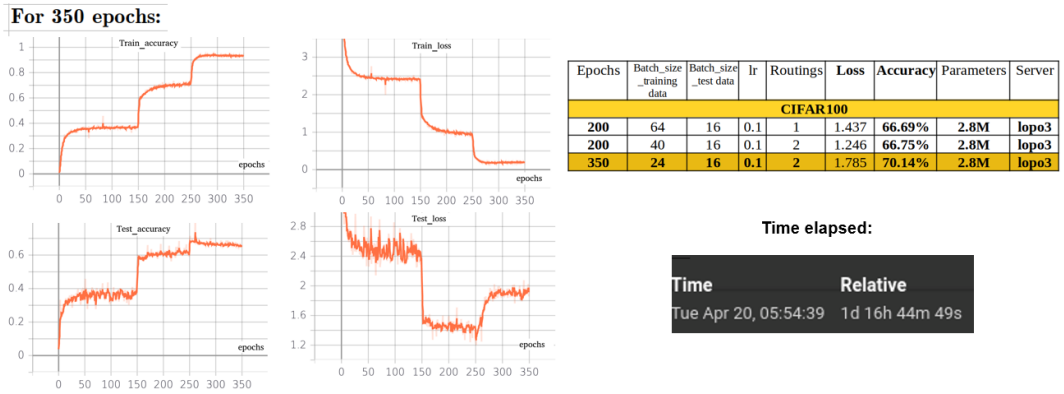
4.4.3 YaoCaps evaluation

This is the implementation of the YaoCaps on CIFAR100 dataset. The input layer has size by 32x32x3. In this case, several attempts have been performed. The best result is obtained when the model is trained for 350 epochs: the batch size for training data is equal to 24, while the batch size for testing data is 16. Moreover, the number of parameters of the model is about 2.8 million and the number of routing process is 2. The learning rate starts from 0.1 and decreases from 150 to 250 epochs. It is important to point out that in this case there is only the encoder part.

Model
Input layer: (input_shape=(32,32,3))
Backbone: ResNet blocks
PrimaryCaps: (num_caps=32, caps_dim=36, kernel_size=1, strides=1, padding='same')
Convolutional_Capsules **Capsules1:** (num_caps=32, caps_dim=36, kernel_size=3, strides=2) **Capsules2:** (num_caps=32, caps_dim=36, kernel_size=3, strides=1) **FCLayer:** (num_caps=20, caps_dim=36)
Class_Capsules: (caps_dim= 36, num_caps=n_classes=100)

Figure 4.17: Implementation of the YaoCaps on CIFAR100

According to the configuration written in Figure 4.17, the best test accuracy obtained is 70.14%, as shown in Figure 4.18. Moreover, in all plots there are two steps, when the learning rate changes.



Epochs	Batch_size _training data	Batch_size _test data	lr	Routings	Loss	Accuracy	Parameters	Server
CIFAR100								
200	64	16	0.1	1	1.437	**66.69%**	**2.8M**	**lopo3**
200	40	16	0.1	2	1.246	**66.75%**	**2.8M**	**lopo3**
350	**24**	**16**	**0.1**	**2**	1.785	**70.14%**	**2.8M**	**lopo3**

Time elapsed:

Time	Relative
Tue Apr 20, 05:54:39	1d 16h 44m 49s

Figure 4.18: YaoCaps on CIFAR100: Plots and results

4.4.4 FMCaps evaluation

This is the implementation of the FMCaps on CIFAR100 dataset. The input layer has size by 32x32x3. In this case, several attempts have been performed. The best result is obtained when the model is trained for 200 epochs: the batch size is equal to 128. Moreover, the number of parameters of the model is about 0.93 million. The learning rate starts from 0.1 and decreases from 81 to 122 epochs. It is important to point out that in this case, as in YaoCaps, there is only the encoder part.

Model
Input layer: (input_shape=(32,32,3))
Backbone: 25x ResNet blocks
PrimaryCaps: (groups=4, atoms=16, kernel_size=5, strides=2, padding='same') + BatchNormalization
MultiCapsules CapsLayer1: CapsuleTransformDense(num_capsule=32) + BatchNormalization + Activation + LastFMPool CapsLayer2: CapsuleTransformDense(num_capsule=16) + BatchNormalization + Activation + LastFMPool CapsLayer3: CapsuleTransformDense(num_capsule=n_classes=100)+ BatchNormalization + Activation + LastFMPool

Figure 4.19: Implementation of the FMCaps on CIFAR100

According to the configuration written in Figure 4.19, the best test accuracy obtained is 69.44%, as shown in Figure 4.20. Moreover, in all plots there is a big step, when the learning rate changes.

FM Caps	Epochs	Batch size	lr	Loss	Accuracy	Parameters	Server
CIFAR100							
	50	128	0.1	2.57	**49.87%**	**930 k**	**lopo3**
	200	128	0.1	1.69	**69.44%**	**930 k**	**lopo3**
	300	128	0.1	1.73	**69.17%**	**930 k**	**lopo3**

For 200 epochs:

```
Elapsed time: 6191.0711 seconds
Hours, minutes, seconds :  [1, 43, 11]
```

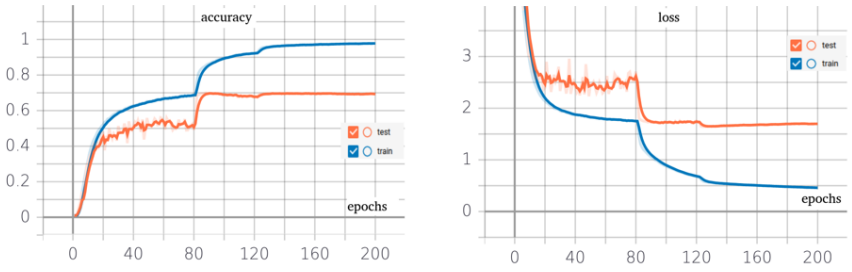


Figure 4.20: FMCaps on CIFAR100: Plots and results

4.5 CINIC10

The CINIC10 (Figure 4.21) is the novel dataset which is not so commonly used in the DL field. In 2018 Darlow [48] proposed this new kind of dataset: it is obtained through a combination of images from **CIFAR10**[31] and **ImageNet32**[49].

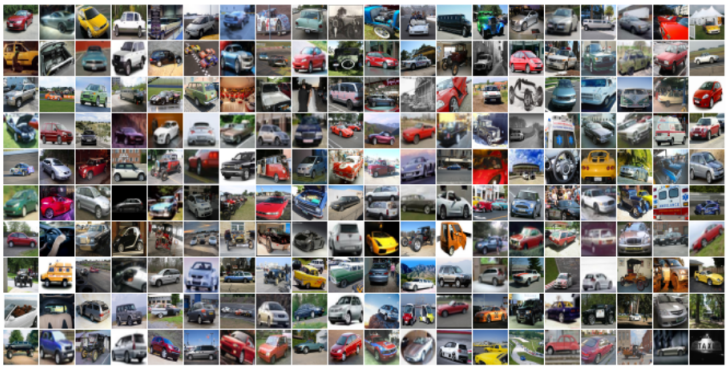


Figure 4.21: CINIC-10 Dataset [50]

The CINIC10 contains 270k images splitted into three equal-sized train, validation, test subsets. In this case the dataset is splitted according to the rule 50/50: the models are trained on the 90.000 training images and evaluated on the 90.000 validation images. It consists of 10 classes as in the MNIST, SVHN and CIFAR10 cases. The input image size of CINIC10 is 32-by-32 RGB pixels.
Darlow et al.[48] evaluated the CINIC10 dataset on several CNNs architectures. The challenge can be evaluating the dataset using CapsNet models in order to make a comparison. Indeed, we evaluate the CINIC10 on two CapsNets architectures: i)Shallow Caps and ii)DeepCaps.

4.5.1 ShallowCaps evaluation

This is the implementation of the ShallowCaps on CINIC10 dataset. The input layer has size by 32x32x3. The model is trained for 50 epochs using the batch size equal to 100; the number of parameters of the model is about 11.7 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. The decoder is composed by three FC layers, useful to reconstruct the images.

According to the configuration written in Figure 4.22, the average test accuracy obtained is 56.72% $\pm$ 1.41, as shown in Figure 4.23.

Model
Input layer: (input_shape=(32,32,3))
Conv2D: (filters=256, kernel_size=9, strides=1, padding='valid', activation='relu')
PrimaryCaps: (dim_capsule=8, n_channels=32, kernel_size=9, strides=2, padding='valid')
DigitCaps: (num_capsule=n_classes=10, dim_capsule=16)
Mask: digitCaps=10*16=160
Decoder: Dense(512, activation='relu', input_dim=16*10)) Dense(1024, activation='relu')) Dense(np.prod(32,32,3), activation='sigmoid')) Reshape(target_shape=(32,32,3))

Figure 4.22: Implementation of the ShallowCaps on CINIC10

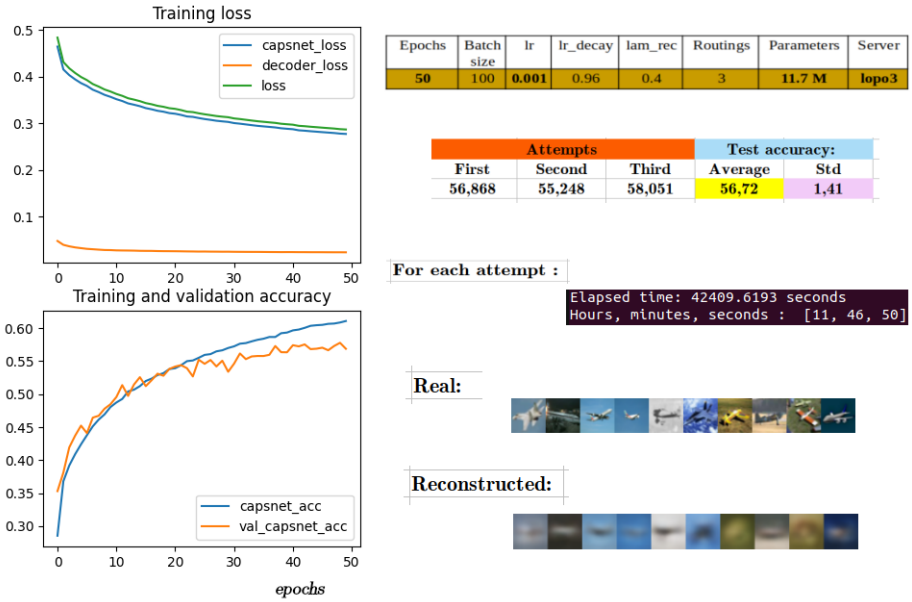


Figure 4.23: ShallowCaps on CINIC10: Plots and results

4.5.2 DeepCaps evaluation

This is the implementation of the DeepCaps on CINIC10 dataset. The input layer has size by 32x32x3: it is resized to 64x64x3 that allows us to go down deeper. The model is trained for 50 epochs using the batch size equal to 64; the number of parameters of the model is about 13.4 million. The learning rate starts from 0.0001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by five Deconvolution 2D layers, useful to reconstruct the images.

According to the configuration written in Figure 4.24, the average test accuracy obtained is 75.24% $\pm$ 0.33, as shown in Figure 4.25.

Input Layer:	32x32x3	Resize:	64x64x3

Model

Input layer: (input_shape=(64,64,3))

Conv2D: (filters=128, kernel_size=3, strides=1, padding='same', activation='relu') + BatchNormalization

4 CapsCells

-1x
- Conv2DCaps: (n_capsules=32, n_neurons=4, kernel_size=3, strides=2)
- 3x Conv2DCaps: (n_capsules=32, n_neurons=4, kernel_size=3, strides=1)

-2x
- Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=2)
- 3x Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=1)

-Dynamic Routing 3D
- Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=2)
- ConvCapsuleLayer3D:(kernel_size=3, num_capsule=32, num_atoms=8, strides=1, padding='same', routings=3)
- 2x Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=1)

DigitCaps: (num_capsule=n_classes=10, dim_capsule=32)

Mask_CID: (digitCaps)

Decoder: Dense(input_dim=32, activation="relu", output_dim=8 * 8 * 16)) + Reshape + BatchNormalization
Deconvolution2D(nb_filters=64, nb_row=3, nb_col=3, subsample=1)
Deconvolution2D(nb_filters=32, nb_row=3, nb_col=3, subsample=2)
Deconvolution2D(nb_filters=16, nb_row=3, nb_col=3, subsample=2)
Deconvolution2D(nb_filters=8, nb_row=3, nb_col=3, subsample=2)
Deconvolution2D(nb_filters=3, nb_row=3, nb_col=3, subsample=1) + Activation("relu"))
Reshape(target_shape=(64, 64, 3))

Figure 4.24: Implementation of the DeepCaps on CINIC10

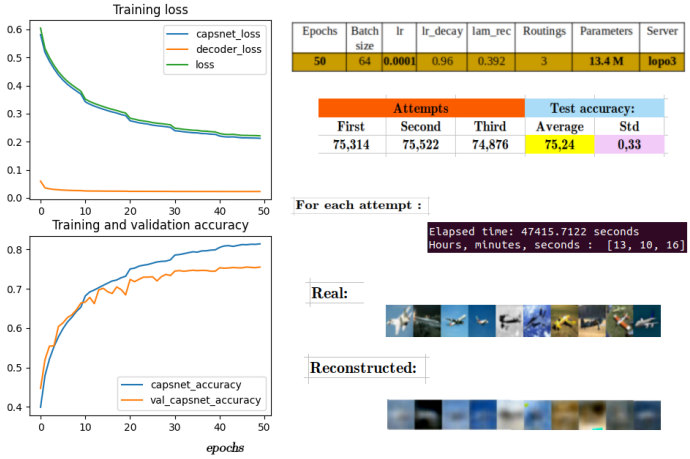


Figure 4.25: DeepCaps on CINIC10: Plots and results

4.6 Remarks

At the end of chapter, it is important to make some considerations. For all cases, DeepCaps performs better than ShallowCaps. In particular, considering SVHN and CINIC10 datasets, DeepCaps reaches 96.48% and 75.24%, respectively, while ShallowCaps gets 94.96% and 56.72%, respectively. In Figure 4.26 there is a histogram showing these results.

Furthermore, a comparison between CIFAR10 and CIFAR100 is done. Yao Caps achieves the best results in CIFAR10 (94.70%) and CIFAR100 (70.14%), as shown in Figure 4.27.

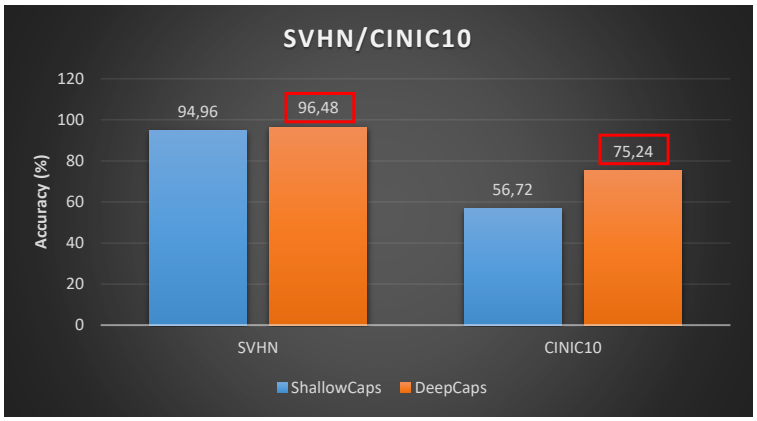


Figure 4.26: Results on SVHN and CINIC10

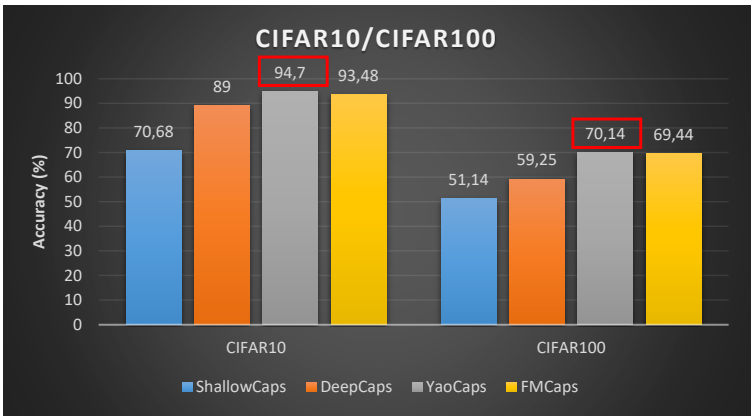


Figure 4.27: Results on CIFAR10 and CIFAR100

Chapter 5

High resolution datasets: MLRS and COVID19

In the previous chapter some simulations were carried out on some known datasets. Following the same procedure used previously, we evaluate how the CapsNets behave with some new datasets such as MLRS and COVID19. Up to now, we have done simulations on datasets that contain images with size 32x32. In the case of the new datasets the images are of **high resolution**: for MLRS dataset the image size is 256x256 while for COVID19 it is 299x299.

Having high resolution datasets involves some implementation problems [1] which affect the memory usage. They can be summarized in two points:

- **Loading process** : it depends on the way you write the code to load the data and the image processing technique applied.
- **Convolutional process** : it involves operations between tensors which entail a certain computational cost.

The mechanism involved is the following: we load images and they become an input volume for models (like CNNs and CapsNets). The input image is transformed into tensors and a classification is performed through the convolution process. If the image has a large size, the tensors become large and consequently the computational cost of the convolutional process becomes high: in this way there is not enough space in the memory. Figure 5.1 shows a graphical summary of the conceptual flow.

[1] In all simulations we used only one GPU.

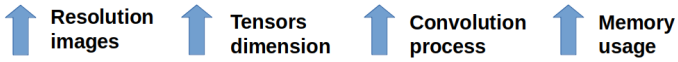


Figure 5.1: High resolution issue: there is not enough space in the memory

One solution would be to resize the images to a smaller size: for example from 256x256 to 32x32. Furthermore, by applying the "Trial & Error procedure", the best model can be found by modifying the CapsNet parameters (number of capsules, capsule size, first convolutional layers ...) so that the memory can better manage every computational cost. Figure 5.2 shows an example of image resizing.

Figure 5.2: Solution: resize the image

The previous works on MLRS [51] and on COVID19 [52] were done using CNNs architectures.

What is the accuracy using CapsNets?

Indeed, we evaluate both the datasets on two CapsNets architectures: i)Shallow Caps and ii)DeepCaps.

5.1 Remote sensing dataset: MLRS Evaluation

At the end of 2020, Xiaoman Qi et al. [51] introduced the MLRSNet dataset [53] (Figure 5.3) in order to improve image recognition techniques in **remote sensing**[2].

The MLRS contains 109.161 high-resolution optical satellite images allowing to capture different perspectives of the Earth. According to the "85/15 rule" we splitted all the images obtaining 92.786 training images and 16.375 validation images. Furthermore, the number of classes is 46 and each category contains from 1,500 to 3,000 images.

[2]Remote sensing is a process that allows to detect and photograph the physical characteristics of the Earth such as rivers, lakes, mountains through the use of cameras on satellites and airplanes.

Figure 5.3: MLRS Dataset [51]

5.1.1 ShallowCaps evaluation

This is the implementation of the ShallowCaps on MLRS dataset. The input layer has size by 256x256x3: it is resized with shorter size 32x32x3. The model is trained for 50 epochs using the batch size equal to 32; the number of parameters of the model is about 23.8 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. The decoder is composed by three FC layers, useful to reconstruct the images.

Input Layer:	256x256x3	Resize:	32x32x3

Model
Input layer: (input_shape=(32,32,3))
Conv2D: (filters=256, kernel_size=9, strides=1, padding='valid', activation='relu')
PrimaryCaps: (dim_capsule=8, n_channels=32, kernel_size=14, strides=2, padding='valid')
DigitCaps: (num_capsule=n_classes=46, dim_capsule=16)
Mask: digitCaps=46*16=736
Decoder: Dense(512, activation='relu', input_dim=16*46)) Dense(1024, activation='relu')) Dense(np.prod(32,32,3), activation='sigmoid')) Reshape(target_shape=(32,32,3))

Figure 5.4: Implementation of the ShallowCaps on MLRS

According to the configuration written in Figure 5.4, the average test accuracy obtained is 72.714% $\pm$ 0.30, as shown in Figure 5.5.

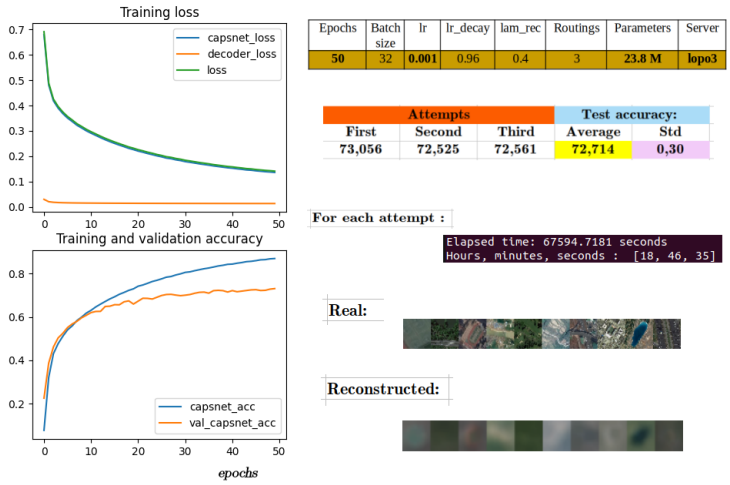


Figure 5.5: ShallowCaps on MLRS: Plots and results

5.1.2 DeepCaps evaluation

This is the implementation of the DeepCaps on MLRS dataset. The input layer has size by 256x256x3: it is resized to 32x32x3. The model is trained for 50 epochs using the batch size equal to 32; the number of parameters of the model is about 14.4 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by four Deconvolution 2D layers, and not five as it happens in the others configurations.

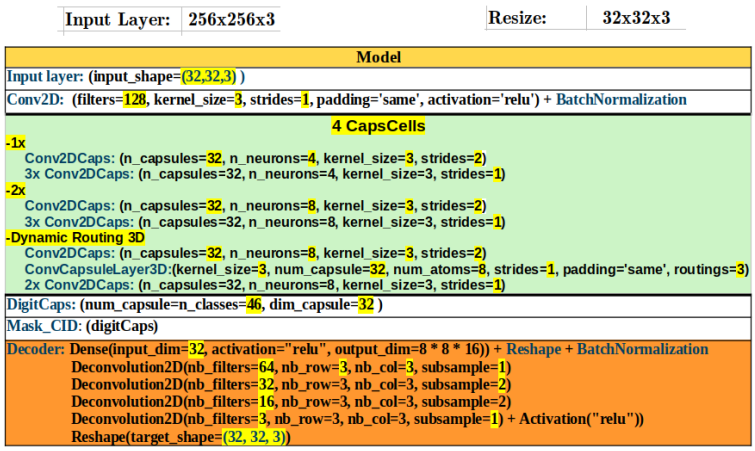


Figure 5.6: Implementation of the DeepCaps on MLRS

According to the configuration written in Figure 5.6, the average test accuracy obtained is 82.23% ± 1.60, as shown in Figure 5.7.

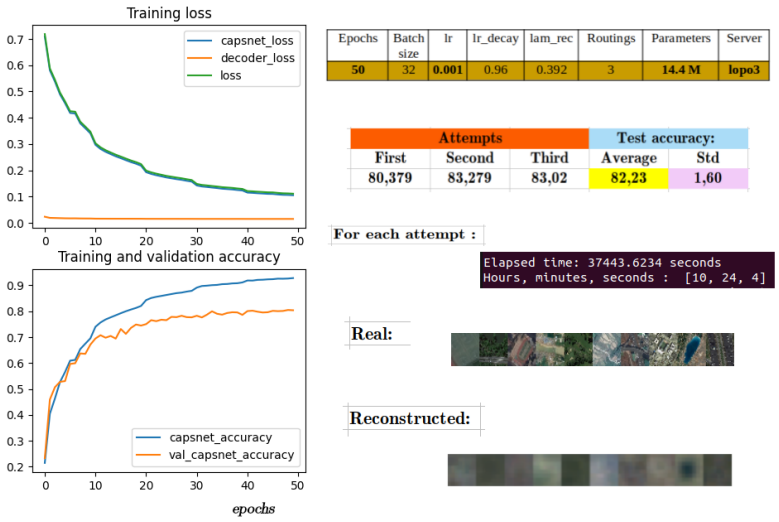


Figure 5.7: DeepCaps on MLRS: Plots and results

5.2 Medical images dataset: COVID 19

At the end of 2019, a new virus came to change our life and our habits: the **COVID-19**. It is an infectious respiratory disease caused by the coronavirus **SARS-CoV-2**.[54] The COVID19 is transmitted via droplets of saliva and leads to serious consequences, such as respiratory failure and in some cases even death. In this context, the medical diagnosis[3] becomes fundamental to recognize the virus and prevent it from spreading further. The simplest diagnostic method is to perform an **Chest X-ray** [4] to see if there is any damage to the lungs. Nowadays, medical diagnostics based on **image classification** are spread among diagnostic techniques: it is useful as supplement medical decision making.

Indeed, researchers have collected a multitude of **Chest X-ray** images in order to improve diagnostic techniques through deep learning. At the end of March 2021, Xiao Qi et al.[52] proposed a new dataset, called **COVID-Ti Dataset**.[55]

[3]Medical diagnosis is a process that allows us to know which disease are involved to the person's symptoms.

[4]Chest X-ray is a non invasive medical test commonly used in pneumology field: the chest is flooded with ionizing radiation allowing to create images of the inside the body in order to see both the chest and related organs such as the heart and lungs.

Nowdays, it is the largest dataset of COVID19 disease (Figure 5.8) and it is constantly updated: many researchers are helping to extend it further.

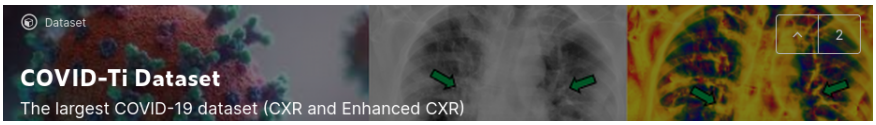
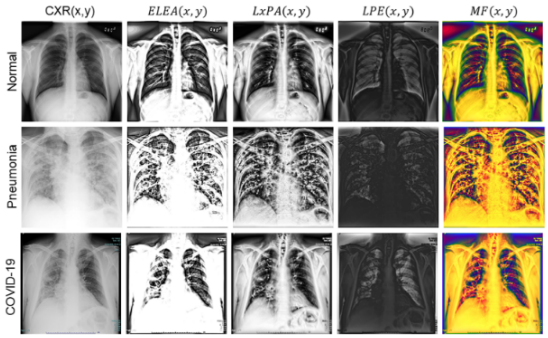


Figure 5.8: COVID19 Dataset [55]

This big dataset is splitted into two datasets:

- **COVID19_CXR**: where the Chest X-Ray images are in a grey-scale configuration (in Figure 5.9 CXR images)
- **COVID19_ENH**: where the Chest X-ray images are enhanced using a **image enhancement method** [5] based on local phase. In this way the images are in RGB configuration. (in Figure 5.9 MF images)

Each dataset contains three classes : **"Normal"**, **"Pneumonia"** and **"COVID19"**. Indeed, the DL models have to recognize if the image belongs to one of these classes.



Normal	8.851
Pneumonia	6.045
COVID19	3.795

Figure 5.9: Chest X-ray images: CXR and enhanced CXR (MF) [52]

According to steps introduced in the previous chapter, we decide to split each dataset according to the rule 85/15. In this way both COVID19 CXR and COVID19 ENH contain 15.887 training images and 2.804 validation images. Furthermore, the images are not uniformly distributed: each class has a different number of image as it can seen in the table above.

Xiao Qi et al. [52] produced some **CNNs simulations** on the COVID19 dataset: they splitted both COVID19 CXR and COVID19 ENH into "training

[5] In image processing field, the image enhancement is a technique that improves the quality of digital images making easier to identify key features.

data", "validation data" and "test data", according to a certain "rate rule" that is different from the rule adopted in the thesis. He demonstrated that the simulations on the enhanced CXR images (COVID19 ENH) give better results compared to the classical CXR ones (COVID19 CXR).
The challenge can be evaluating both COVID19 CXR dataset and COVID19 ENH on CapsNets architectures in order understand if there is the same behaviour as in the CNNs models.

5.3 COVID 19 CXR

5.3.1 ShallowCaps evaluation

First configuration:

At the beginning we propose a ShallowCaps model applied to COVID19 CXR following the same procedure as in previous cases.
The input layer has size by 299x299x1: it is resized with shorter size 32x32x1. The training dataset follows the rule 85/15. The model is trained for 100 epochs using the batch size equal to 100. The learning rate starts from 0.0001 and it is reduced by half after each 20 epochs until the end. The decoder is composed by three FC layers, useful to reconstruct the images.

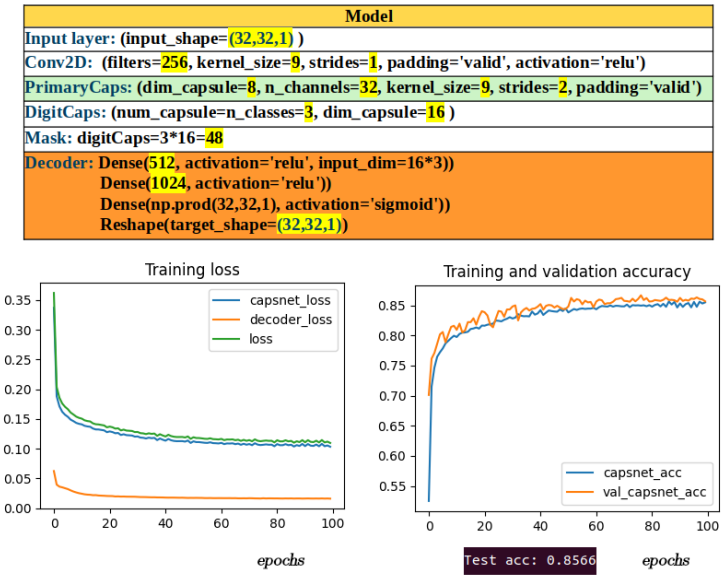


Figure 5.10: First configuration of the ShallowCaps on COVID19 CXR: model configuration and results.

In Figure 5.10 it is possible to highlight that both training and validation curves fit data with a similar behavior. In particular, it seems that the curve of validation capsnet accuracy is slightly higher than the curve of training capsnet accuracy for all the time simulation. It is a problem because it seems that the training process is not so good in this way. Indeed, the test accuracy is only 85.66 %: it is less than expected. According to what is declared in the previous chapter, the "rule of thumb" should be fulfilled. We need to change the train and the validation curves improving the performance in order to generalize well the configuration.

How to fulfill the "rule of thumb"?

This issue can be solved through several strategies. A solution can be found following these two steps:

i. Apply a regularization method adding BN layer after first convolutional layer

ii. Change the splitting rate datasets increasing the test dataset **(rule 50/50)**

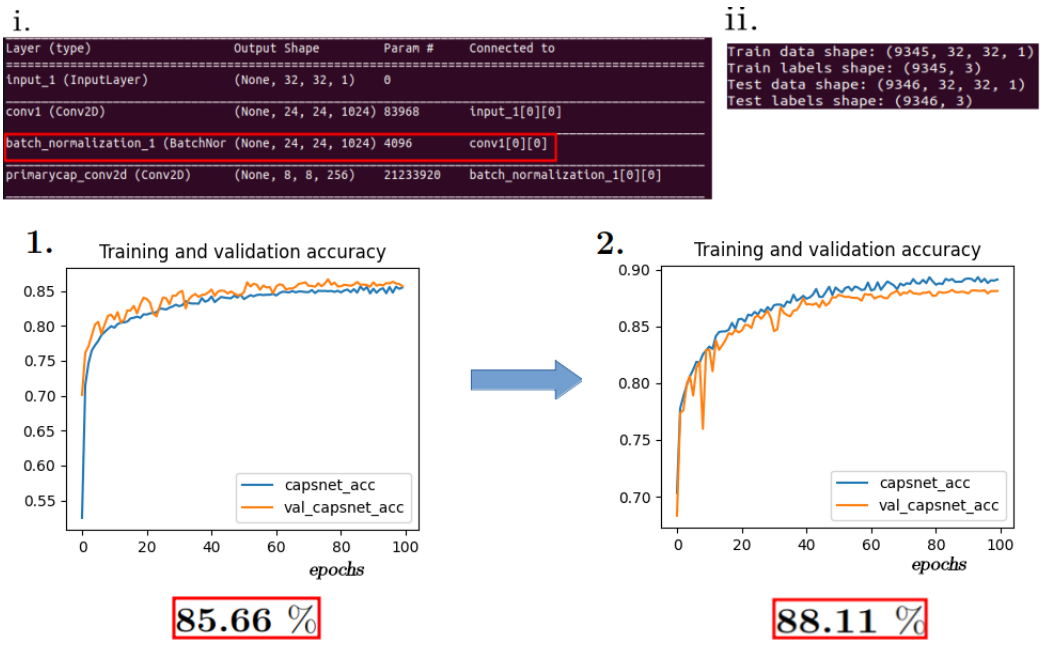


Figure 5.11: Solution for the issue found: add a BN layer after the first Conv2D and change the splitting rate into 50/50. Note the change in the curves (from plot "1" to plot "2").

Thanks to the strategy described above, the training images go from 15.887 to 9.345 and the validation images go from 2.804 to 9.346. Furthermore, the test

accuracy goes from 85.66 % to 88.11 %. In Figure 5.11 it is appreciable to note the change in the curves (from plot "1" to plot "2"): after the solution proposed, the curve of validation capsnet accuracy is less than the curve of training capsnet accuracy for all the time simulation.

ShallowCaps final configuration:

Going into depth, this is the final implementation of the ShallowCaps on COVID 19 CXR. The configuration of the model is similar as before, with the introduction of the BN layer and the splitting rate dataset is 50/50. The number of parameters of the model is about 23.7 million.

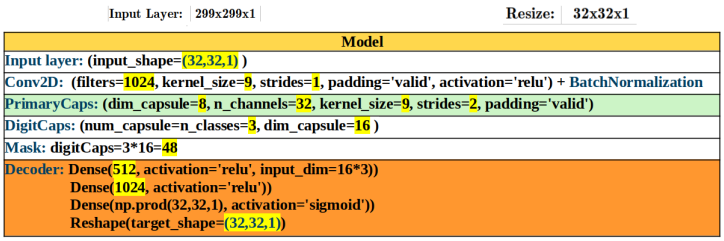


Figure 5.12: Implementation of the ShallowCaps on COVID19 CXR : final configuration

According to the configuration written in Figure 5.12, the average test accuracy obtained is 88.14% ± 0.07, as shown in Figure 5.13.

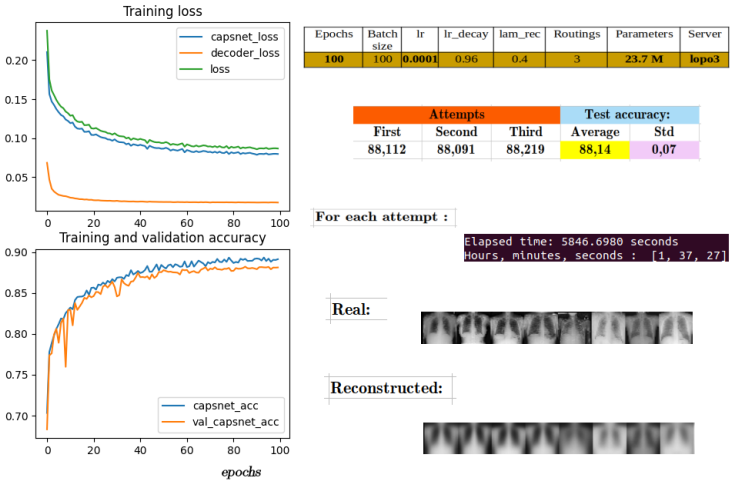


Figure 5.13: ShallowCaps on COVID19 CXR: Plots and results

5.3.2 DeepCaps evaluation

This is the implementation of the DeepCaps on COVID19 CXR dataset. The input layer has size by 299x299x1: it is resized to 64x64x1 that allows us to go down deeper. The model is trained for 100 epochs using the batch size equal to 64; the number of parameters of the model is about 8.8 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by five Deconvolution 2D layers, useful to reconstruct the images.

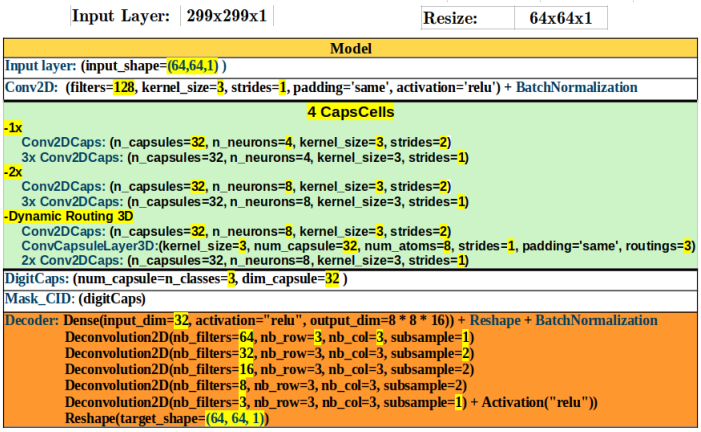


Figure 5.14: Implementation of the DeepCaps on COVID19 CXR

According to the configuration written in Figure 5.14, the average test accuracy obtained is 93.34% ± 0.16, as shown in Figure 5.15.

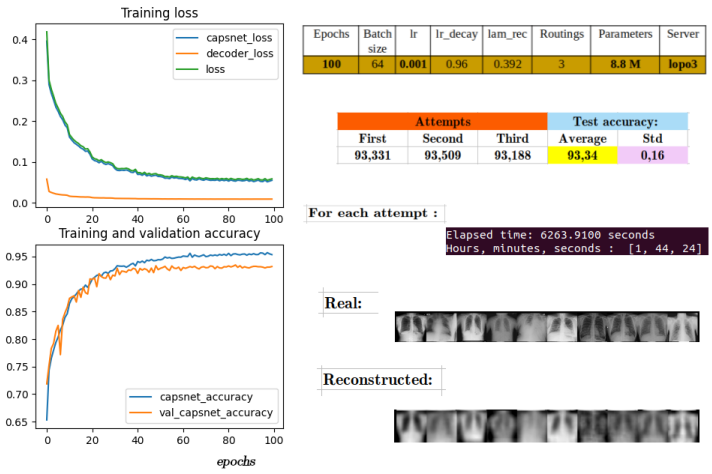


Figure 5.15: DeepCaps on COVID19 CXR: Plots and results

5.4 COVID 19 ENH

5.4.1 ShallowCaps evaluation

This is the implementation of the ShallowCaps on COVID 19 ENH dataset. The input layer has size by 299x299x3: it is resized with shorter size 32x32x3. The training dataset follows the rule 85/15: it is not changed as in COVID19 CXR. The model is trained for 100 epochs using the batch size equal to 100; the number of parameters of the model is about 25.9 million. We adopt a BN after the Conv2D layer, but the final result is similar also without the BN: it is important to remember that the BN is fundamental in the COVID19 CXR case. The learning rate starts from 0.0001 and it is reduced by half after each 20 epochs until the end. The decoder is composed by three FC layers, useful to reconstruct the images.

Input Layer:	299x299x3	Resize:	32x32x3

Model
Input layer: (input_shape=(32,32,3))
Conv2D: (filters=1024, kernel_size=9, strides=1, padding='valid', activation='relu') + BatchNormalization
PrimaryCaps: (dim_capsule=8, n_channels=32, kernel_size=9, strides=2, padding='valid')
DigitCaps: (num_capsule=n_classes=3, dim_capsule=16)
Mask: digitCaps=3*16=48
Decoder: Dense(512, activation='relu', input_dim=16*3)) Dense(1024, activation='relu')) Dense(np.prod(32,32,3), activation='sigmoid')) Reshape(target_shape=(32,32,3))

Figure 5.16: Implementation of the ShallowCaps on COVID19 ENH

According to the configuration written in Figure 5.16, the average test accuracy obtained is 93.10% $\pm$ 0.11, as shown in Figure 5.17.

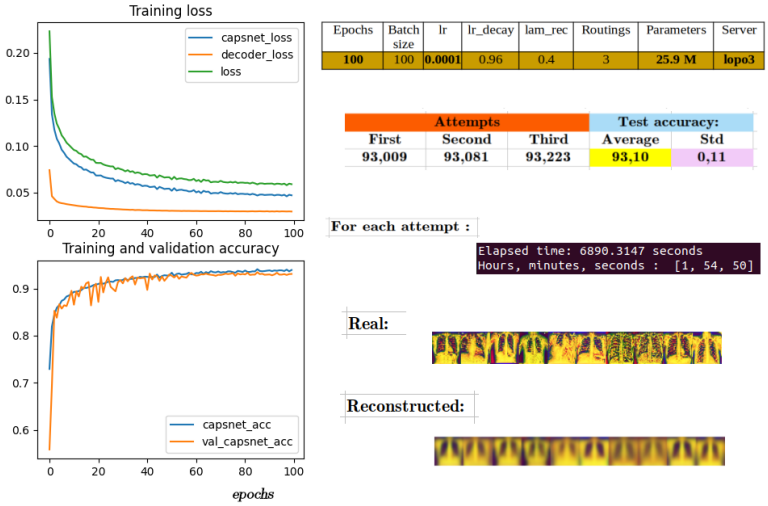


Figure 5.17: ShallowCaps on COVID19 ENH: Plots and results

5.4.2 DeepCaps evaluation

This is the implementation of the DeepCaps on COVID19 ENH dataset. The input layer has size by 299x299x3: it is resized to 64x64x3 that allows us to go down deeper. The model is trained for 100 epochs using the batch size equal to 64; the number of parameters of the model is about 8.8 million. The learning rate starts from 0.001 and it is reduced by half after each 20 epochs until the end. We adopt a batch normalization(BN) after the Conv2D layer. The decoder is composed by five Deconvolution 2D layers, useful to reconstruct the images.

```
Input Layer:  299x299x3                    Resize:  64x64x3
Model
Input layer: (input_shape=(64,64,3) )
Conv2D:  (filters=128, kernel_size=3, strides=1, padding='same', activation='relu') + BatchNormalization
4 CapsCells
-1x
   Conv2DCaps: (n_capsules=32, n_neurons=4, kernel_size=3, strides=2)
   3x Conv2DCaps: (n_capsules=32, n_neurons=4, kernel_size=3, strides=1)
-2x
   Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=2)
   3x Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=1)
-Dynamic Routing 3D
   Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=2)
   ConvCapsuleLayer3D:(kernel_size=3, num_capsule=32, num_atoms=8, strides=1, padding='same', routings=3)
   2x Conv2DCaps: (n_capsules=32, n_neurons=8, kernel_size=3, strides=1)
DigitCaps: (num_capsule=n_classes=3, dim_capsule=32 )
Mask_CID: (digitCaps)
Decoder: Dense(input_dim=32, activation="relu", output_dim=8 * 8 * 16)) + Reshape + BatchNormalization
         Deconvolution2D(nb_filters=64, nb_row=3, nb_col=3, subsample=1)
         Deconvolution2D(nb_filters=32, nb_row=3, nb_col=3, subsample=2)
         Deconvolution2D(nb_filters=16, nb_row=3, nb_col=3, subsample=2)
         Deconvolution2D(nb_filters=8, nb_row=3, nb_col=3, subsample=2)
         Deconvolution2D(nb_filters=3, nb_row=3, nb_col=3, subsample=1) + Activation("relu"))
         Reshape(target_shape=(64, 64, 3))
```

Figure 5.18: Implementation of the DeepCaps on COVID19 ENH

According to the configuration written in Figure 5.18, the average test accuracy obtained is 94.01% ± 0.13, as shown in Figure 5.19.

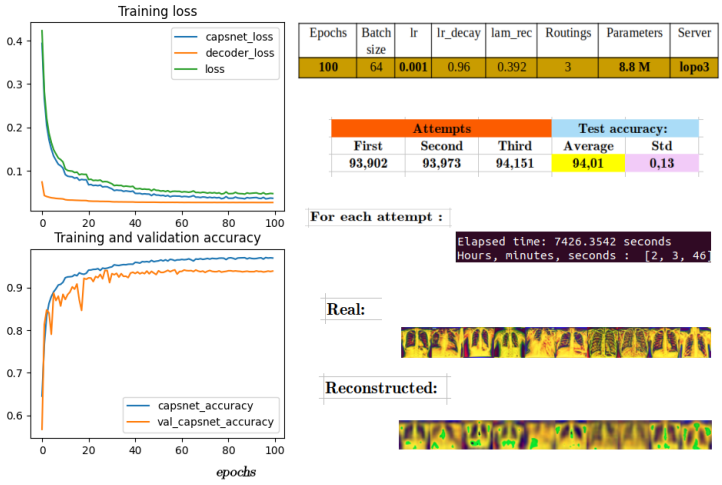


Figure 5.19: DeepCaps on COVID19 ENH: Plots and results

5.5 Remarks

Finally, it is possible to make a comparison between the results obtained using ShallowCaps and DeepCaps and other implementations. In particular, we decided to run simulations using ResNet50.

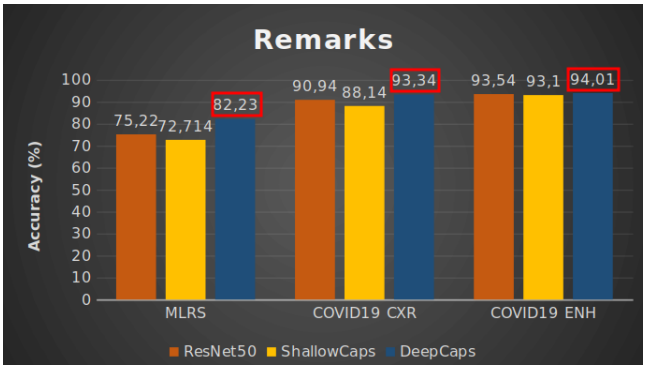


Figure 5.20: Remarks

In Figure 5.20, it is evident that the DeepCaps outperforms the SOTA of ResNet50 for all datasets. Indeed:

- **MLRS:** the DeepCaps achieves **82.23** % while the accuracy of ResNet50 is **75.22** %;
- **COVID19 CXR:** the DeepCaps reaches **93.34** % while the accuracy of ResNet50 is **90.94** %;
- **COVID19 ENH:** the DeepCaps gets **94.01** % while the accuracy of ResNet50 is **93.54** %.

Part III

CapsNet robustness

Chapter 6

White Box configuration: Gradient based attacks

In the Chap. 2 the adversarial examples are introduced. Now, it's time to go into depth with these concepts. In particular, there will be an analysis of the common gradient based attacks in white box setting. After that, it is important to understand how the adversarial attacks can be crafted in ResNet and CapsNet cases. Finally, in the CapsNet context there is another way to mislead the models: the Vote Attack. This chapter will introduce all these arguments, while the simulations will be performed in the next one.

6.1 Gradient based attacks

In the Chap.2 the taxonomy of adversarial attacks is explained. In this section, we will discuss the **white box** case where the adversary has full access to the model: architecture and weights are known.

In literature there are two types of **gradient based methods:**

- **One step method:** it performs one iteration to generate adversarial examples.
- **Iterative steps methods:** iteratively apply perturbations multiple times with a small step size.

6.1.1 Fast Gradient Sign Method (FGSM)

The **FGSM** is the first example of "one step method" which is introduced by Goodfellow et al. (2014) [56] . FGSM is a fast method which allows to craft a

perturbation adding a small vector η that takes into account the sign of the gradient of the loss function with respect to the input x following the formula:

$$\eta = \epsilon \cdot sign(\nabla_x J(\theta, x, y)) \tag{6.1}$$

where ϵ is a small perturbation, x is the input, J is the loss function and θ are the parameters of the model.

$$x' = x + \eta = x + \epsilon \cdot sign(\nabla_x J(\theta, x, y)) \tag{6.2}$$

It is important to highlight that the generation of adversarial examples depends on the gradient of the loss function J (used to train the DNNs) computed in the backpropagation process. In Figure 6.1 there is a graphical representation of the FGSM attack.

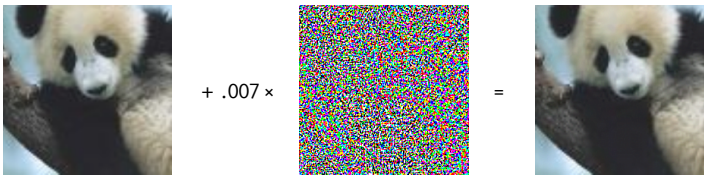


Figure 6.1: Fast Gradient Sign Method [56]

6.1.2 Basic Iterative Method (BIM)

Kurakin et. al. (2016) [57] introduced the **Basic Iterative Method** (BIM) which extends the FGSM idea through the application of an iterative optimizer: in this case the small perturbation is not added with a single step, but iteratively applying multiple steps. After each step, the pixel values of the intermediate results are clipped by updating the x ' so that the neighbourhood constraint is satisfied.

$$x'_{t+1} = Clip\{x'_t + \alpha \cdot sign(\nabla_x J(\theta, x'_t, y))\} \tag{6.3}$$

6.1.3 Momentum Iterative Method (MIM)

Y. Dong et al. (2018) [58] proposed a **Momentum Iterative Method** (MIM) which is a BIM process with the introduction of a momentum optimizer. In practise BIM has an important drawback: at each iteration, the adversarial example follows the direction of the sign of the gradient causing poor local maxima. In MIM the solution is to accelerate the GD algorithms accumulating in a velocity vector the gradient direction of the cost function for each step. In this way the update

directions are stabilized and the poor local maxima is escaped. The gradient g_{t+1} is updated by the introduction of a decay factor $\mu > 0$:

$$g_{t+1} = \mu \cdot g_t + \frac{\nabla_x J(x'_t, y)}{\|\nabla_x J(x'_t, y)\|_1} \tag{6.4}$$

If the decay factor is equal to 0, the MIM becomes BIM. The update rule of x_{t+1} is similar to the BIM case :

$$x'_{t+1} = Clip\{x'_t + \alpha \cdot sign(g_{t+1})\} \tag{6.5}$$

6.1.4 Projected Gradient Descent (PGD)

Madry et al. (2017) [59] introduced a **Projected Gradient Descent** (PGD): it's similar to BIM, but in this case PGD projects the adversarial samples (from each iteration) into the ϵ-ball of x.

$$x'_{t+1} = Proj\{x'_t + \alpha \cdot sign(\nabla_x J(\theta, x'_t, y))\} \tag{6.6}$$

6.2 Implementation

Until now, several adversarial attacks are presented.

How can we produce an adversarial attack, in practice?

Various libraries help us to craft an adversarial attack: **Cleverhans** [60] and **Foolbox** [61] are used in this dissertation. [1] For other details it is possible to consult Cleverhans [63] and Foolbox [64] documentations.
The adversarial attack is created following some steps:

i. Train the model using clean images and save weights about it

ii. Load the model trained

iii. Transform the model, written in Keras or PyTorch, into the model useful for Cleverhans or Foolbox through a **Wrapper**. In this way, the toolbox has any informations about the model. [2]

[1] Nowdays, another toolbox is becoming popular : **AdverTorch** [62]. It is built on PyTorch and allows to implement various attacks and defences, in a similar way as in the previous toolboxes.

[2] In this way, the toolbox (Cleverhans or Foolbox) computes gradients on its own. For further explanations it is possible to consult Carlini's articles [65], [66].

iv. Craft adversarial attacks applying the attack functions thanks to the toolboxes modules

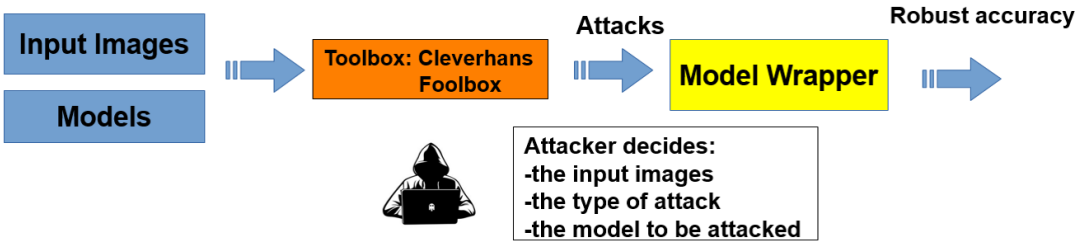


Figure 6.2: Adversarial attacks implementation

Figure 6.2 shows that the input images and the model are fed into the toolbox, which transforms the model in its language producing an adversarial attack to the "Model Wrapper". The **robust accuracy** is computed : it is different from the accuracy at "clean setting" because it takes into account that the attacks is crafted. In this context, the attacker decides the input volume, the attack and the model to be attacked. In this chapter the attack is performed using **ResNet**, **CapsNet** models.

6.2.1 ResNet Attacks

In ResNet setting, the output final layer is attacked as it is shown in Figure 6.3.

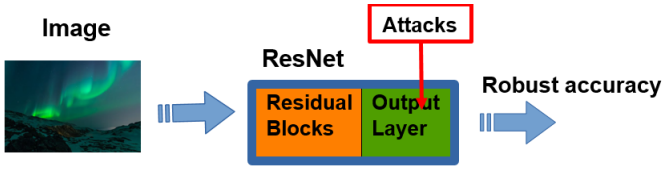


Figure 6.3: ResNet Attacks

6.2.2 Caps Attacks

In CapsNet setting, we can split the Caps Attacks into two configurations:

- **ShallowCaps Attacks**
- **DeepCaps Attacks**

In both cases, the output capsules are attacked directly following the formula :

$$\delta^* = \arg\max \mathcal{H}(Z(x+\delta), y) = \mathcal{L}(softmax(Z(x+\delta), y)) \tag{6.7}$$

6.2.3 Vote Attacks

Jindong Gu et al. [47] proposed a new way to craft adversarial attack in CapsNet setting, as shown in Figure 6.4.

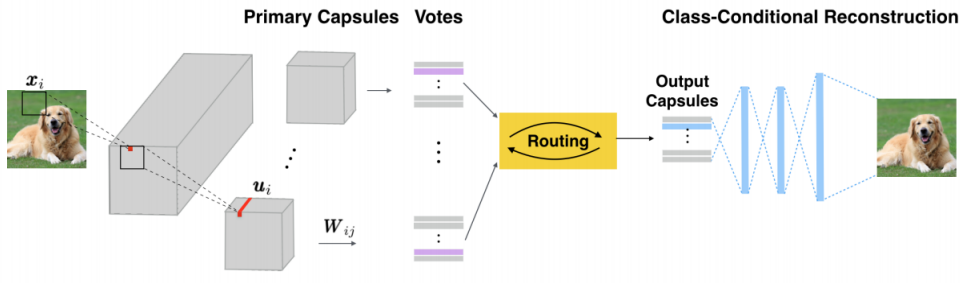


Figure 6.4: Caps Attacks & Vote Attack [47]

In this case, the attack is done considering the **votes** before the routing process, instead of the output capsules.

$$\delta^* = \arg\max \mathcal{H}(log(g(\frac{1}{N}\sum_{i=1}^{N}(f_v)^i(x+\delta))), y) \tag{6.8}$$

6.3 Remarks

There are other attacks to remark in the literature. For example, three can be mentioned:

- **DeepFool** (2016): Moosavi et al. [67] proposed an iterative adversarial attack where the idea is to find the closest distance from the original input to the decision boundary [68] of adversarial examples
- **Carlini** & **Wagner** (2016): it is a strong adversarial attack [69]
- **DDN** (2019):it induces misclassification with L2 norm, by decoupling the direction and the norm of the adversarial perturbation that is added to the image [70]

Chapter 7

Gradient based results

After the introduction of the main concepts in the previous chapter, it's time to run several simulations in order to analyze the results obtained. In particular, there will be an evaluation of CapsNet robustness against adversarial attacks both in Medical images and Remote sensing datasets. It should be noted that they are new datasets: it is the **first time** that these types of simulations are performed on them. After that, in the CapsNet context, the impact of Vote Attack is performed on more complex dataset (CIFAR100). [1]

7.1 Adversarial attacks on Medical Images

In the context of Medical Images, X. Ma (2020) [71] proposed a deeper understanding of the CNNs robustness against adversarial attacks taking into account Fundoscopy, Chest X-ray and Dermoscopy datasets useful for diagnosing diabetic retinopathy, lung diseases and skin cancer respectively.
According to the idea declared previously, the CapsNet robustness against adversarial attacks (FGSM, PGD, BIM, MIM) is performed on Medical images for **COVID19 CXR** and **COVID19 ENH** cases. These datasets are different from the previous ones because they are specific to consider the "COVID19 class" as lung disease too. For each dataset, the analysis is done in **white box untargeted** setting, varying the size of perturbation ϵ. It is performed following two steps:

- Compare all adversarial attacks in ResNet50 and DeepCaps cases understanding what is the best attack

[1] For Medical Images and Remote sensing cases, the implementation is done through Keras & TensorFlow framework with the Cleverhans toolbox. In CIFAR100, the framework is PyTorch while the toolbox is Foolbox.

- Compare the robustness of ResNet50 and DeepCaps for each attack

7.1.1 COVID19 CXR

i. **ResNet50:**
At the beginning PGD,BIM and MIM have the same behavior. PGD becomes stronger than others when $\epsilon = 0.038$, as shown in Figure 7.1 .

ii. **DeepCaps:**
At the beginning BIM and MIM perform better than PGD, but the situation changes when $\epsilon = 0.018$, as shown in Figure 7.1.

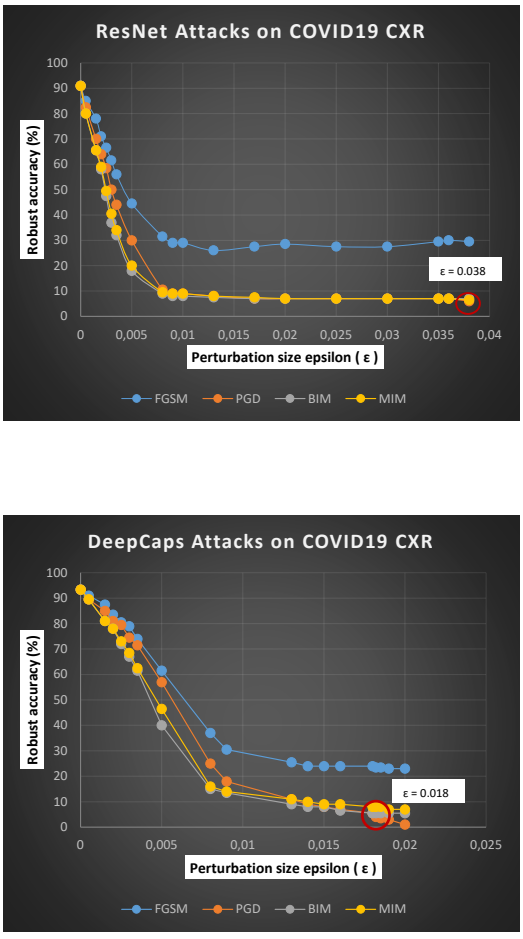


Figure 7.1: ResNet & DeepCaps Attacks on COVID CXR

$\epsilon = 0.08$	**No Attack**	**FGSM**	**PGD**	**BIM**	**MIM**
ResNet50	90.94 %	35.5 %	**0.5** %	1 %	1 %
$\epsilon = 0.0203$	**No Attack**	**FGSM**	**PGD**	**BIM**	**MIM**
DeepCaps	93.34 %	23 %	**0.5** %	5 %	6.5 %

Table 7.1: COVID19 CXR: Evaluation of ResNet and DeepCaps robustness at the end of curves, in the neighbourhood of the values where the robust accuracy goes to zero

In all cases i) and ii) **PGD** is best attack performed, while **FGSM** is the worst one. In particular, in **ResNet case** the robust accuracy of PGD goes to 0.5% when $\epsilon = 0.08$, while the same happens when $\epsilon = 0.0203$ in **DeepCaps case**, as shown in Table 7.1.
The second step is to compare the robustness of ResNet and DeepCaps for each attack.

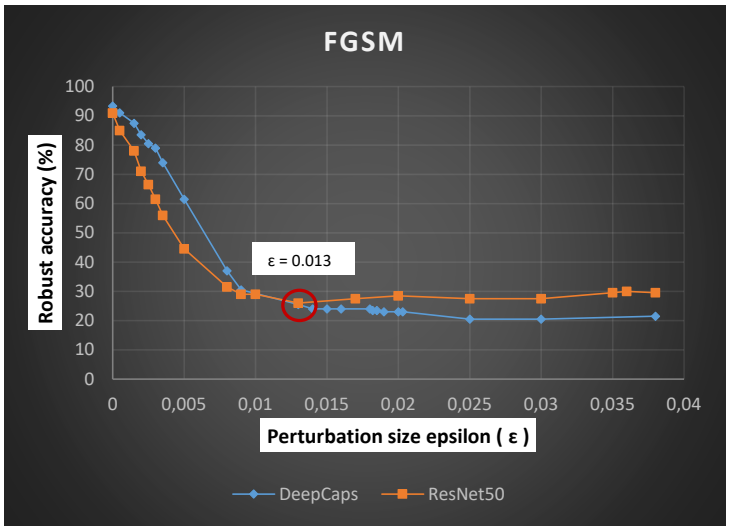


Figure 7.2: FGSM: ResNet & DeepCaps on COVID19 CXR

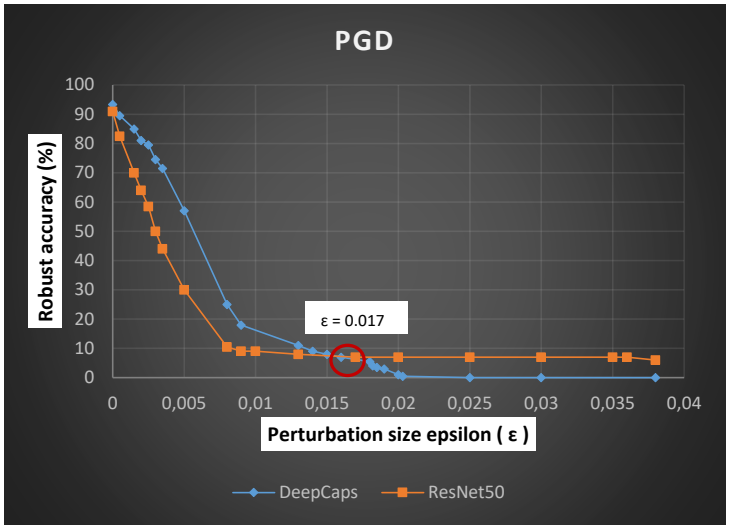


Figure 7.3: PGD: ResNet & DeepCaps on COVID19 CXR

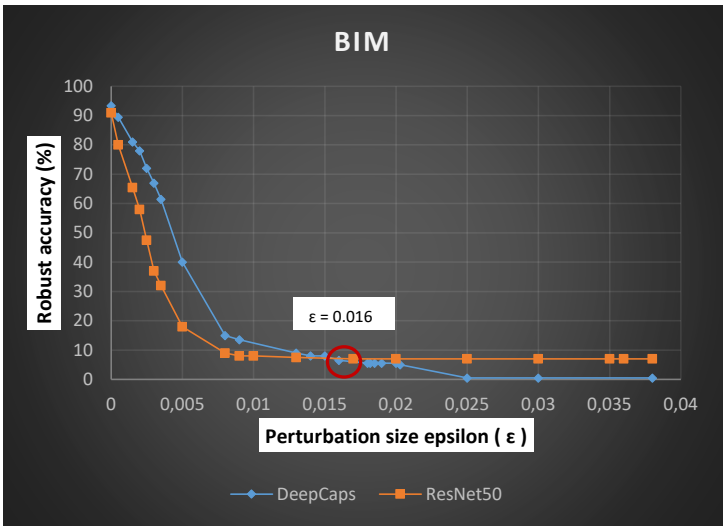


Figure 7.4: BIM: ResNet & DeepCaps on COVID19 CXR

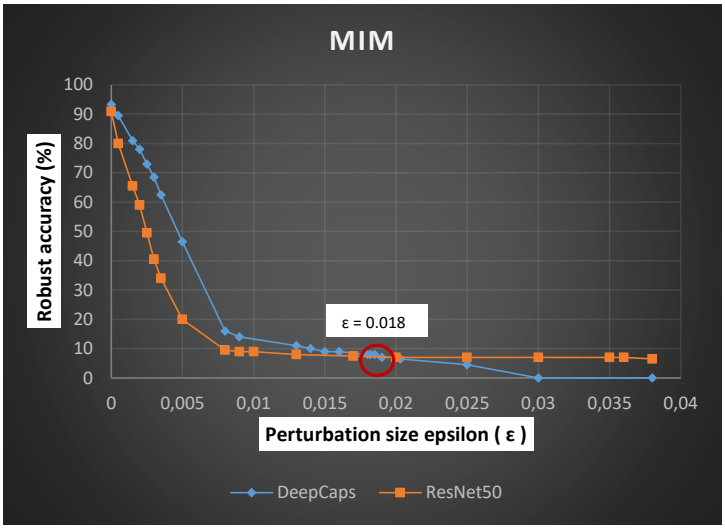


Figure 7.5: MIM: ResNet & DeepCaps on COVID19 CXR

In all cases the behaviour of curves is similar: for **small perturbations** DeepCaps is the most robust, while for **perturbations greater** than a certain ϵ the ResNet50 performs better. In particular, it happens: **FGSM** ($\epsilon > 0.013$, Figure 7.2), **PGD** ($\epsilon > 0.017$, Figure 7.3), **BIM** ($\epsilon > 0.016$, Figure 7.4) and **MIM** ($\epsilon > 0.018$, Figure 7.5).

7.1.2 COVID19 ENH

i. **ResNet50:**
At the beginning PGD,BIM and MIM have the same behavior. PGD becomes stronger when $\epsilon = 0.047$, as shown in Figure 7.6.

ii. **DeepCaps:**
BIM performs better than other attacks for all perturbations ϵ, as shown in Figure 7.6.

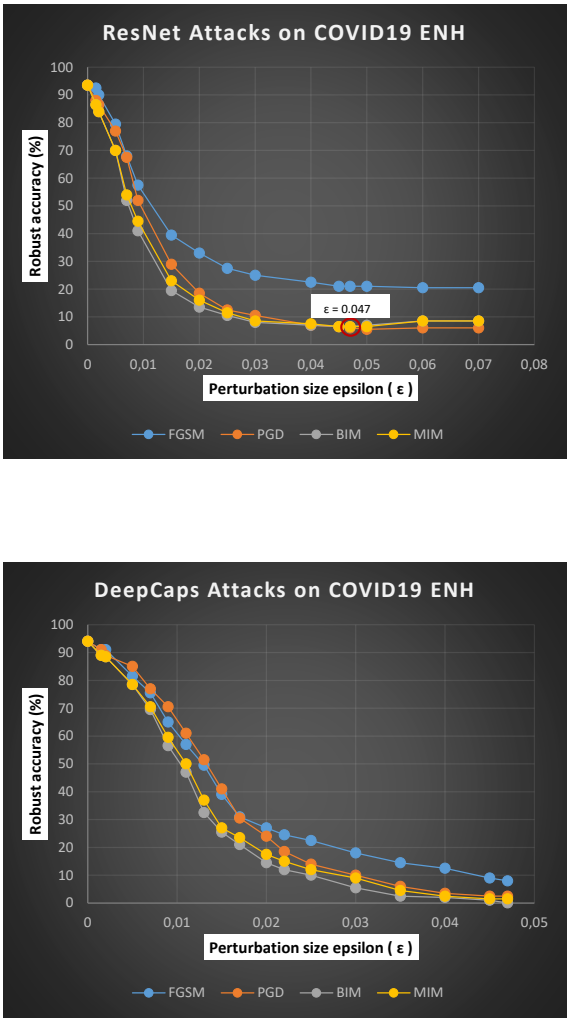


Figure 7.6: ResNet & DeepCaps Attacks on COVID ENH

$\epsilon = 0.3$	**No Attack**	**FGSM**	**PGD**	**BIM**	**MIM**
ResNet50	93.54 %	26 %	**0.5** %	9.5 %	8.5 %
$\epsilon = 0.05$	**No Attack**	**FGSM**	**PGD**	**BIM**	**MIM**
DeepCaps	94.01 %	8 %	1 %	**0.5** %	1 %

Table 7.2: COVID19 ENH: Evaluation of ResNet and DeepCaps robustness at the end of curves, in the neighbourhood of the values where the robust accuracy goes to zero

In all cases i) and ii) **FGSM** is the worst attack performed. In **ResNet case PGD** is the best attack: the robust accuracy goes to 0.5% when $\epsilon = 0.3$. Moreover, **BIM** performs better than others in **DeepCaps case**: the robust accuracy goes to 0.5% when $\epsilon = 0.05$, as shown in Table 7.2 .
The second step is to compare the robustness of ResNet and DeepCaps for each attack.

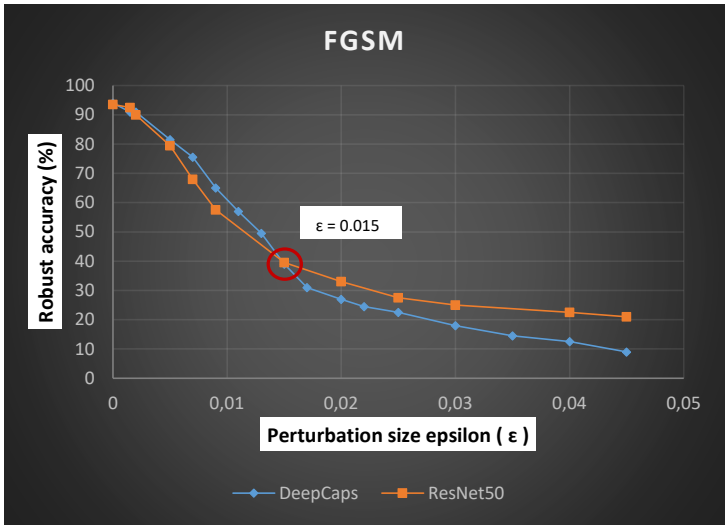


Figure 7.7: FGSM: ResNet & DeepCaps on COVID19 ENH

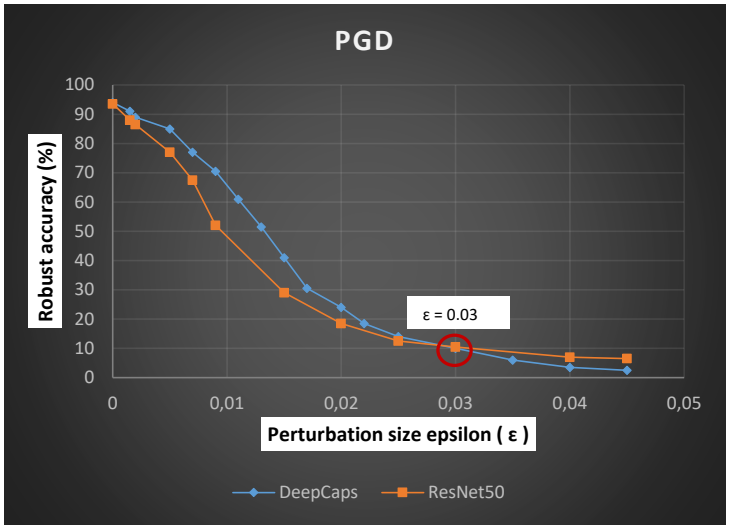


Figure 7.8: PGD: ResNet & DeepCaps on COVID19 ENH

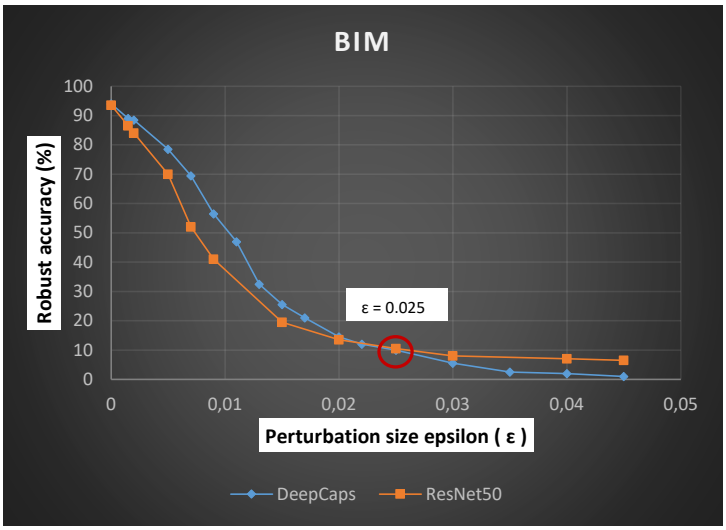


Figure 7.9: BIM: ResNet & DeepCaps on COVID19 ENH

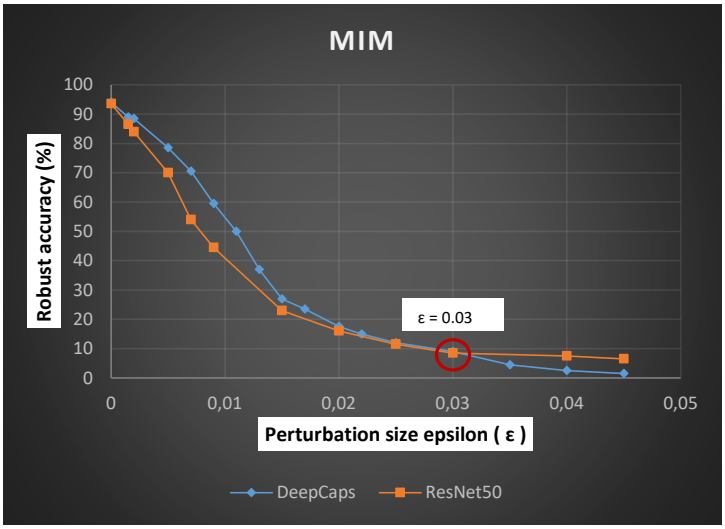


Figure 7.10: MIM: ResNet & DeepCaps on COVID19 ENH

In all cases the behaviour of curves is similar: for **small perturbations** DeepCaps is the most robust, while for **perturbations greater** than a certain ϵ the ResNet50 performs better. In particular it happens: **FGSM** ($\epsilon > 0.015$, Figure 7.7), **PGD** ($\epsilon > 0.03$, Figure 7.8), **BIM** ($\epsilon > 0.025$, Figure 7.9) and **MIM** ($\epsilon > 0.03$, Figure 7.10).
Comparing COVID19 datasets it is straightforward understand that crafting adversarial attacks on COVID19 ENH is more difficult than COVID19 CXR.

7.2 Adversarial attacks on Remote Sensing

The second dataset to be explored is the **MLRS** dataset which belongs to the Remote Sensing field. The approach applied is similar to what it happens in Medical Images scenarios. The goal is to evaluate the robustness of CapsNet against adversarial attacks changing the **pnorm** (l_∞ and l_2). The first step is to compare all adversarial attacks in ResNet50 and DeepCaps cases understanding what is the best attack, while the second step is to compare the robustness of ResNet50 and DeepCaps for attacks with l_∞.

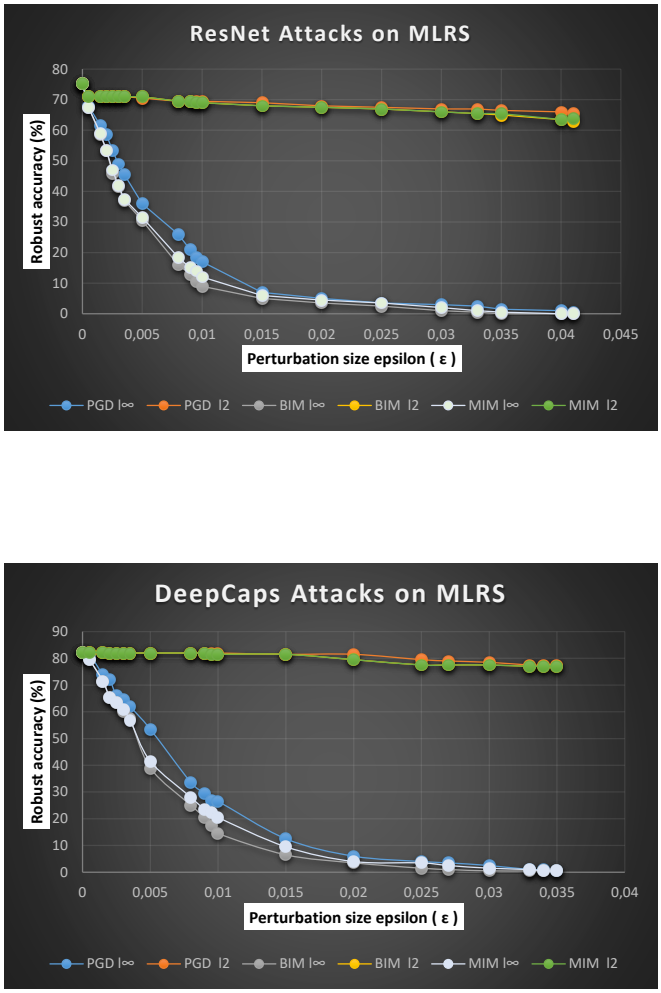


Figure 7.11: ResNet & DeepCaps Attacks on MLRS

Figure 7.11 shows that in all cases adversarial attacks with l_∞ behave better than those with l_2: the l_∞ allows the attack to be stronger. Table 7.3 helps us to make some reflections considering two situations in a random way: for example, **ResNet50** when $\epsilon = 0.033$ and **DeepCaps** when $\epsilon = 0.03$. The robust accuracy is evaluated in both cases.
In **ResNet50** the results demonstrate that PGD l_∞ (2.5 %), BIM l_∞ (0.5 %) and MIM l_∞ (1 %) are stronger than PGD l_2 (67 %), BIM l_2 (65.5 %) and MIM l_2 (65.5 %) respectively.
Moreover, in the case of **DeepCaps** there is the same scenario: PGD l_∞ (2.5 %), BIM l_∞ (0.5 %) and MIM l_∞ (1.5 %) are stronger than PGD l_2 (78.5 %), BIM l_2 (77.5 %) and MIM l_2 (77.5 %) respectively.

	PGD l_∞	PGD l_2	BIM l_∞	BIM l_2	MIM l_∞	MIM l_2
ResNet50						
$\epsilon = 0.033$	2,5 %	67 %	**0,5** %	65,5 %	1 %	65,5 %
$\epsilon = 0.035$	1,5 %	66,5 %	0,0 %	65 %	**0,5** %	65,5 %
$\epsilon = 0.041$	**0,5** %	65,5 %	0,0 %	63 %	0,0 %	64 %
DeepCaps						
$\epsilon = 0.03$	2,5 %	78,5 %	**0,5** %	77,5 %	1,5 %	77,5 %
$\epsilon = 0.034$	1 %	77,5 %	0,5 %	77 %	**0,5** %	77 %
$\epsilon = 0.035$	**0,5** %	77,5 %	0,5 %	77 %	0,5 %	77 %

Table 7.3: ResNet and DeepCaps evaluation against adversarial attacks (l_∞ and l_2) on MLRS dataset

Furthermore, **BIM** l_∞ is best attack performed in all cases. In particular in **ResNet case** the robust accuracy of BIM l_∞ goes to 0.5% when $\epsilon = 0.03$, while the same happens when $\epsilon = 0.033$ in **DeepCaps case**.
The second step is to compare the robustness of ResNet and DeepCaps for each attack with l_∞ norm.

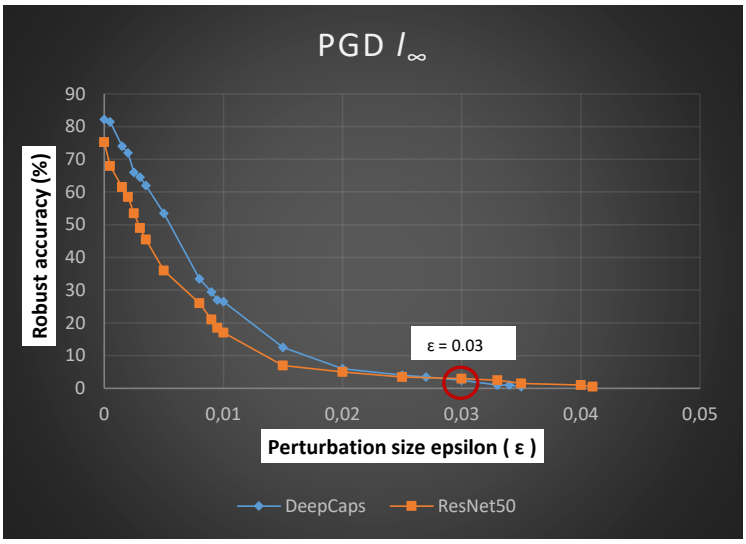


Figure 7.12: PGD l_∞ on MLRS

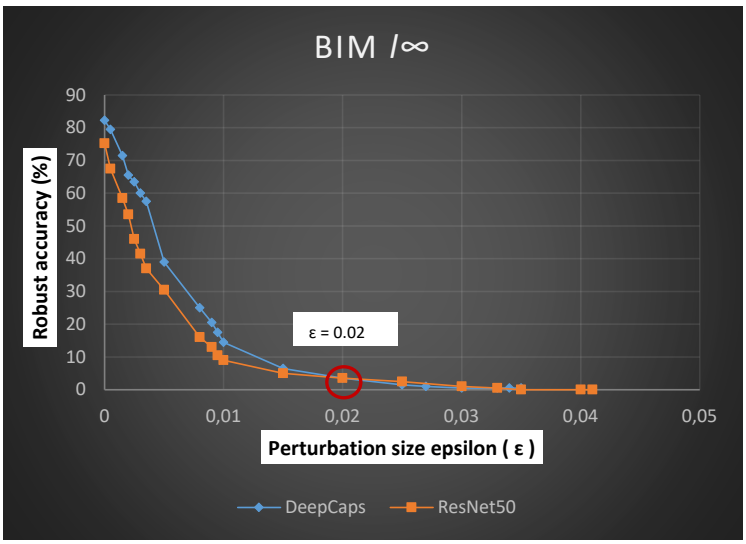


Figure 7.13: BIM l_∞ on MLRS

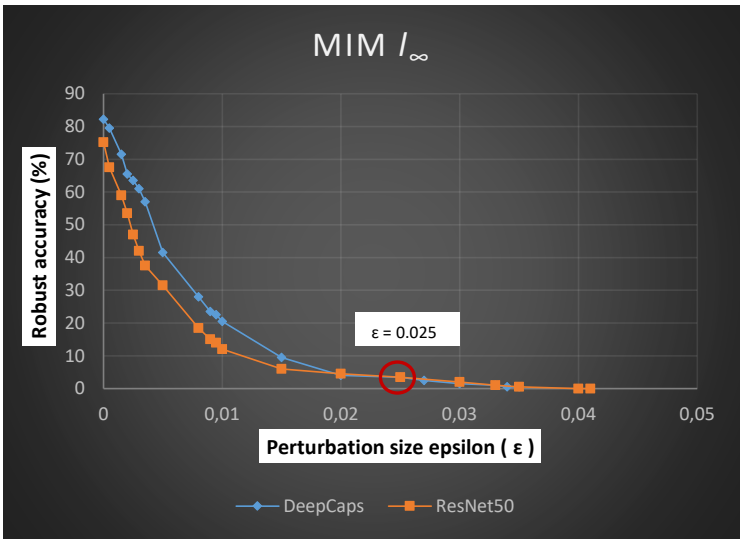


Figure 7.14: MIM l_∞ on MLRS

In all cases the behaviour of curves is similar: **for small perturbations** DeepCaps is the most robust, while for **perturbations greater** than a certain ϵ the ResNet50 performs better. In particular, it happens: **PGD** l_∞ ($\epsilon > 0.03$, Figure 7.12), **BIM** l_∞ ($\epsilon > 0.02$, Figure 7.13) and **MIM** l_∞($\epsilon > 0.025$, Figure 7.14).

7.3 Impact of the Vote Attack on CIFAR100

In a CapsNet configuration, another way to inject attacks is done through Vote Attack, which directly attacks the votes instead of output capsules. Jindong Gu et al. [47] proposed the Vote Attack against popular datasets such as SVHN and CIFAR10. In this sense, the impact of the Vote Attack has been evaluated only on simple datasets until now.

Does the Vote Attack also stronger than Caps Attack in more complex datasets?

The way to answer this question is to analyze the robustness of CapsNet against adversarial attacks on a big dataset like **CIFAR100**. In particular, the robustness is evaluated taking into account **ResNet18** and **ShallowCaps** models. Adversarial attacks are performed in three ways: ResNet attacks, ShallowCaps attacks and

Vote Attacks. The accuracy curves are crafted by FGSM, PGD and BIM with the increasing perturbation size ϵ.

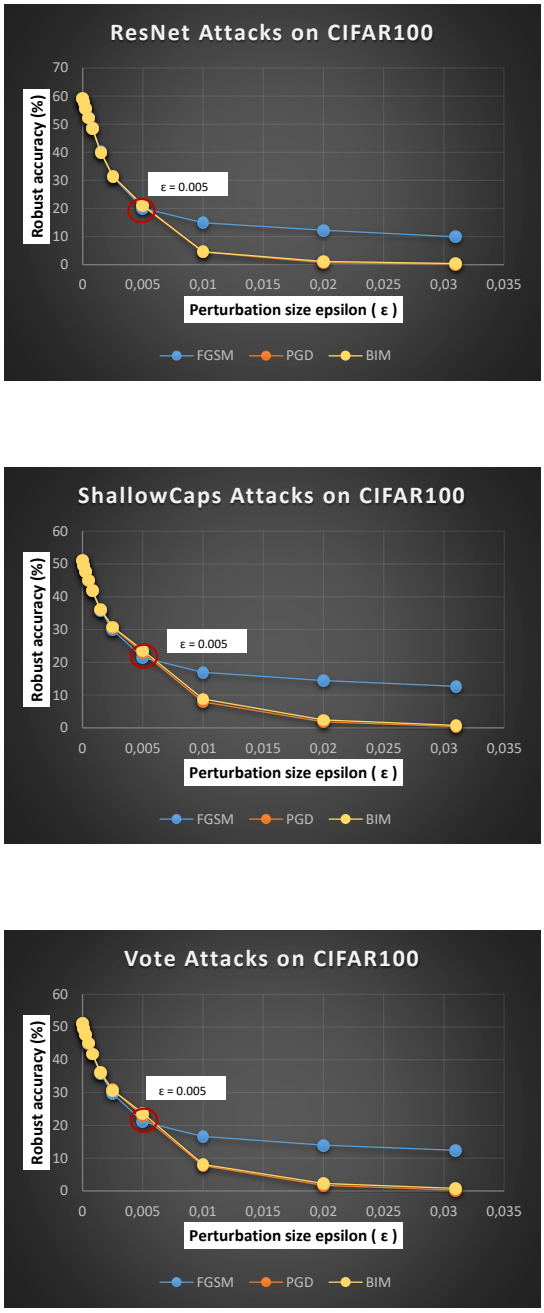


Figure 7.15: ResNet Attacks, ShallowCaps Attacks & Vote Attacks on CIFAR100

In Figure 7.15, at the beginning the behaviour of curves is similar, while after $\epsilon = 0.005$ **FGSM** changes the slope by flattening, while **PGD** and **BIM** continue to decrease with the same trend. It can be seen that FGSM is not powerful compared to PGD and BIM.
Secondly, the robustness of ResNet and ShallowCaps for each attack is evaluated.

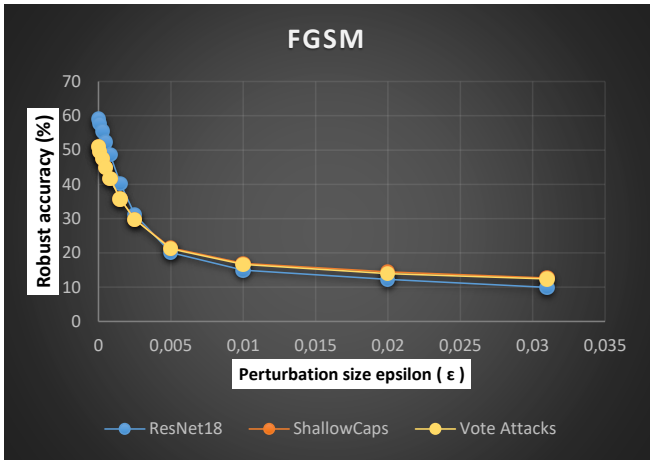


Figure 7.16: FGSM Attacks on CIFAR100

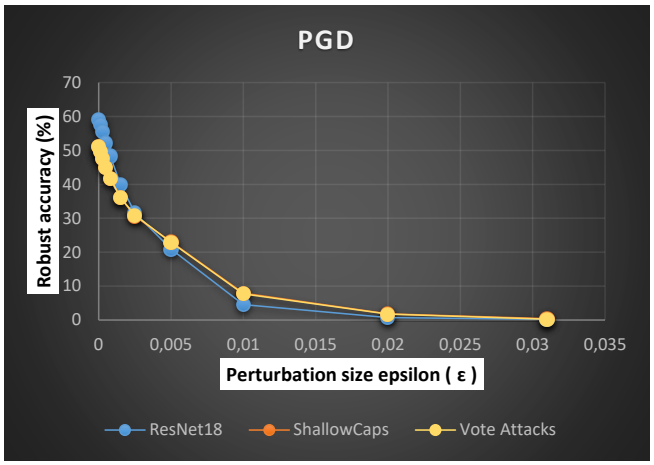


Figure 7.17: PGD Attacks on CIFAR100

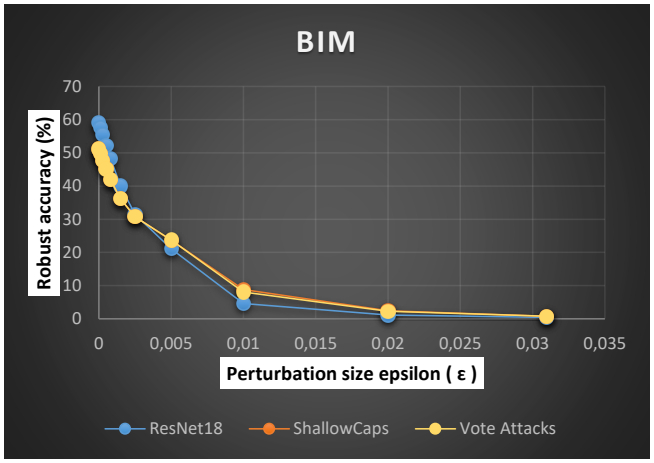


Figure 7.18: BIM Attacks on CIFAR100

$\epsilon = 0.04$	**No Attack**	**FGSM**	**PGD**	**BIM**
ResNet18	59.05 %	8.59 %	0.13 %	0.26 %
ShallowCaps	51.14 %	**11.59** %	**0.19** %	**0.63** %
Vote	51.14 %	**11.27** %	**0.17** %	**0.5** %

Table 7.4: White box attacks on CIFAR100: attacks for $\epsilon = 0.04$

The accuracy curves represent the behavior of the models for each attack. Following the trends in the Figures (7.16, 7.17 and 7.18) and the results of the Table 7.4, two considerations can be obtained in CIFAR100:

- ShallowCaps is more robust than ResNet18 because ShallowCaps resists attacks better
- Vote Attack is stronger than ShallowCaps Attack because robust accuracy values due to Vote Attack are always lower than those caused by Shallow Caps. For example, when $\epsilon = 0.04$, in **Vote** case the robust accuracies of FGSM (11.27 %), PGD (0.17 %) and BIM (0.5 %) are less than FGSM (11.59 %), PGD (0.19 %) and BIM (0.63 %) in **ShallowCaps**.

In this way, it is possible to declare that the Vote Attack **continue** to be stronger than Caps Attack also in more complex dataset like CIFAR100.

Chapter 8

Black Box configuration: Pattern Attack

In the previous chapters many topics have been covered: the evaluation of the performances and the robustness of CapsNet models against adversarial attacks is performed in different fields, from the common benchmarks datasets to the Medical Images and Remote sensing datasets. In this sense, the first aim of the thesis is fulfilled.
This chapter focus on the second goal of this dissertation: design a **novel methodology** that can mislead the CNNs and CapsNet architectures.

8.1 Motivations

The design of the novel attack derives from some considerations that will now be revealed. First of all, our attention went to the concept of **one pixel attack** proposed by N. Narodytska et al. (2016) [72] and Sun et al. (2019) [73], using different algorithms.

In the black box setting, several prior works are done until now. Kurakin et al. (2016) [57] proposed a method that, taking the images from a cell-phone camera, it is possible to craft adversarial examples in the physical world. Moreover, taking into account the high-saliency and low-distortion path, Gragnaniello et al. (2020) [74] introduced an attack that improves the perceptual quality of the adversarial image.

Secondly, several attacks were crafted through the introduction of a new concept: the **adversarial patch**. Brown et al. (2017) [75] generated an **image independent patch** to be placed anywhere inside the original image in order to mislead the models, as shown in Figure 8.1.

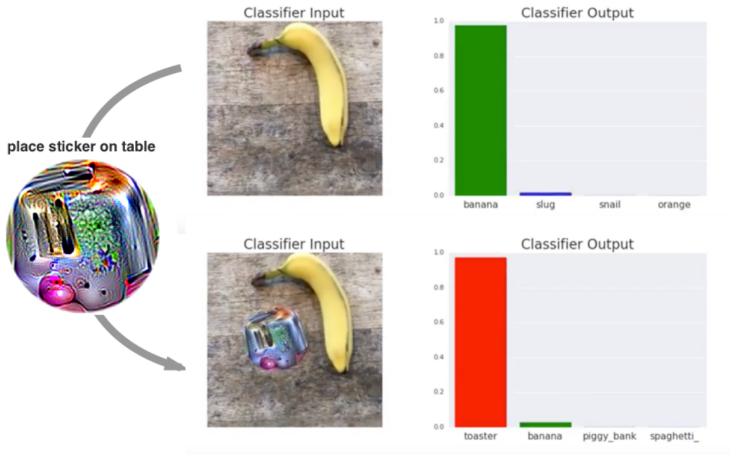


Figure 8.1: Adversarial Patch creates a misclassification: at the beginning the classifier output belongs to class "banana", while after the introduction of a sticker the class changes to "toaster". [75]

Previously, several publications were done: Eykholt et al. (2017) [76] added stickers to road signs in **Traffic sign recognition** field, while Sharif et al. (2016) [77] added glasses to faces in **Face recognition** context, as can be seen in Figure 8.2.

Figure 8.2: At left, there are stickers on road signs [76], while glasses are on face at right [77].

Thirdly, another consideration will be done. In the real world there are several situations where DL models fool the correct classification due to **atmospheric phenomena**. Zhai et al. (2020) [78] simulated various rainy situations using a rain generation process gradient based.

8.2 Pattern Configuration

Taking into account all previous observations, a **novel methodology** is proposed: assuming that the camera lens is dirty due atmospheric conditions (such as rain, snow and hail), it is possible to craft a perturbation changing the pixels of the input following several patterns. Such attack, called **Pattern Attack**, is performed in a **black box setting**:

- the adversary has no informations about network
- only information available is the probability labels
- the attacker can query the network and observe the outcome.

The **Pattern Attack** is performed through the introduction of **drops of water** and **snowflakes** (Figure 8.3).

Figure 8.3: Drops of water [79] & Snowflakes [80]

In reality, a drop of water has a spherical shape while a snowflake has a hexagonal one. For simplicity, a drop or a snowflake can be modelled as a pixel: a small square of the image which has $h \cdot l$ pixels (h = height , l = length).
In this sense, the **Pattern Attack** extends the formulation of **one pixel attack** which is useful for this purpose. Sun et al. (2019) [73] defined the perturbation of a **pixel** as a **tuple** of 5 elements (x, y, r, g, b) where:

- **(x, y)** are the coordinates of the pixel to be modified
- **(r, g, b)** is the color of the pixel in RGB configuration

$$\boxed{pixel_i = (x_i, y_i, r_i, g_i, b_i)} \tag{8.1}$$

In practice, the novel methodology attacks the network , not only with one pixel, but injecting a **specific pattern** following the appropriate algorithm. This action is possible by adding multiple perturbations that imply **more pixels** to modify in the input volume: it is a concatenation of multiple tuples. Figure 8.4 gives a graphical representation of the pixel perturbations.

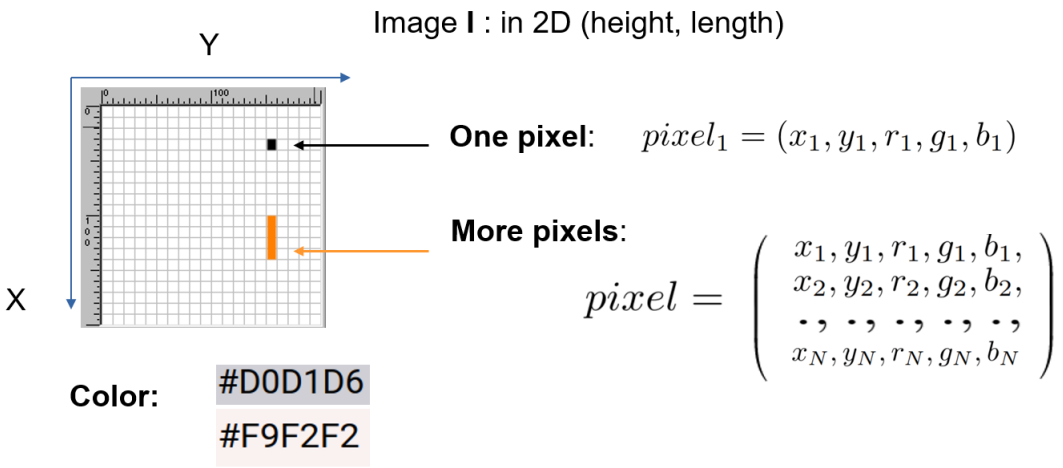


Figure 8.4: Pixel perturbations: a pixel is simply a perturbation of a tuple, while multiple tuples are needed for more pixels

The **color** of the pixel is chosen according to the situation: in a **rain** scenario $color = [208, 209, 214]$ $(\#D0D1D6)$, while for **snow** and **hail** scenarios $color = [249, 242, 242]$ $(\#F9F2F2)$.[1]

How a pattern attack can be generated?

The goal is to create a specific pattern adding **several** drops of water or snowflakes. Using the switch command it is possible to select the attack desired. Pattern Attack (Figure 8.5) is splitted into three attacks: **rain attack**, **snow attack** and **hail attack**. After crafting attack, the adversarial image goes into the black box in a way that there is a comparison between the **prior confidence** in a clean situation and the **new confidence** in a adversarial situation in order to evaluate the **attack success Rate**. ASR measures the success rate of attack taking into account how many times the adversarial attacks mislead the network.

Pattern Attack is applied to **CIFAR10** which has 10 classes: each image has size 32x32x3. Moreover, the novel methodology is evaluated on three models: **LeNet**, **ResNet** and **CapsNet**. The code is written on Keras & TensorFlow framework.

[1]The # symbol followed by an alphanumeric code represents HTML color codes: in this way each type of color is labeled.

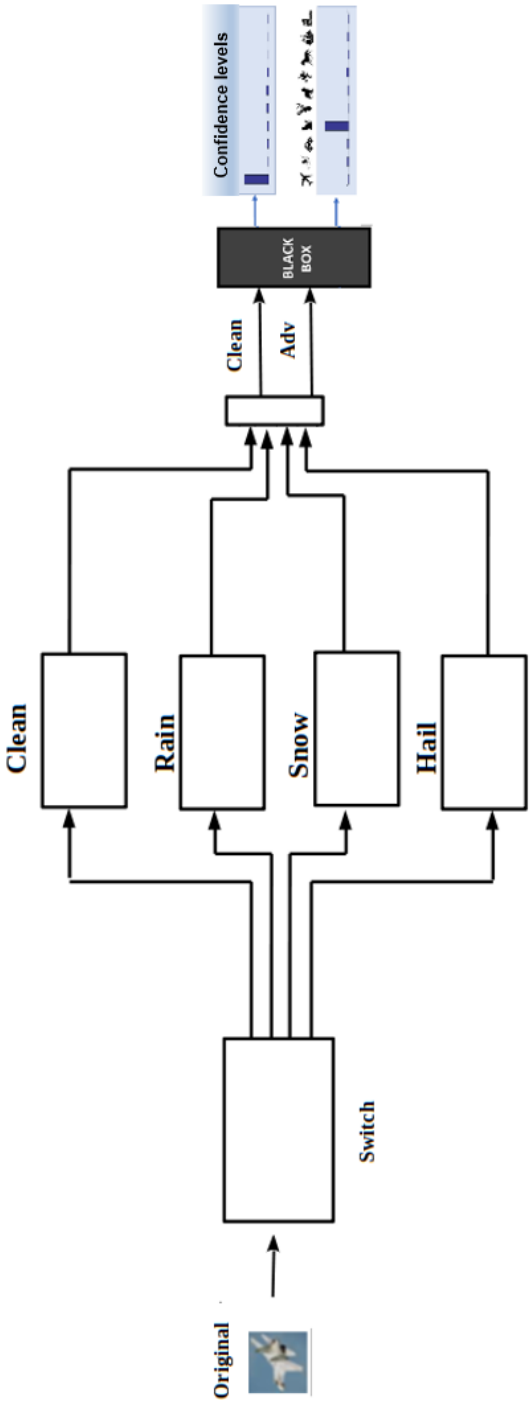


Figure 8.5: Pattern Attack architecture

8.2.1 Rain Attack

Rain attack is the first attack that is based on several drops of water. In the real world the camera lens can be soiled by rain: water droplets make up different patterns, as in Figure 8.6.

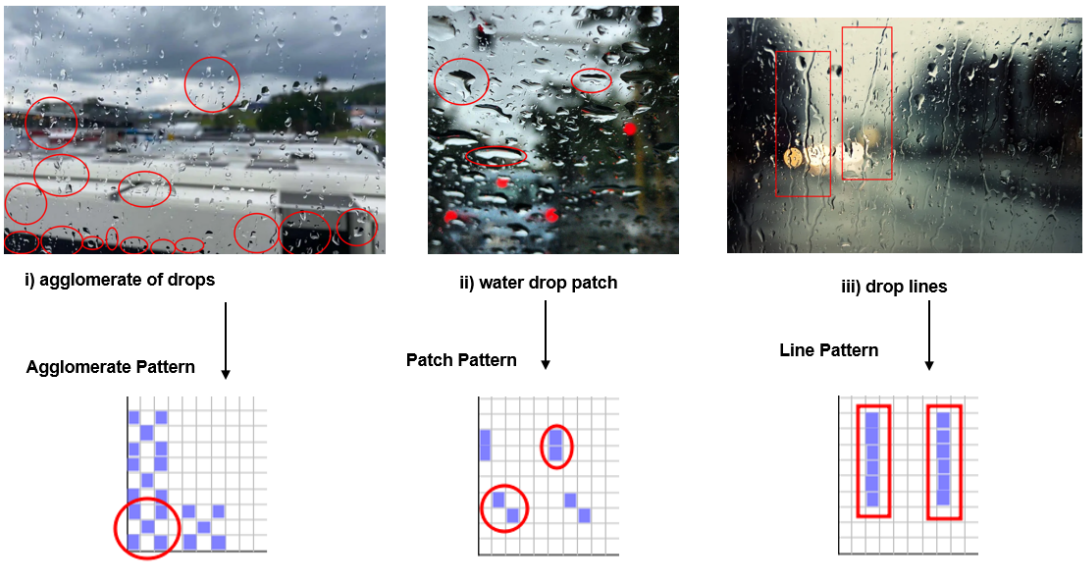


Figure 8.6: Several patterns of drop of waters coming from real environment: i)agglomerate of drops [81], ii)water drop patch [82] and iii) drop lines [83]

The **Agglomerate Pattern** can be modelled as 5 pixels put together following the representation in Figure 8.7.

Algorithm 1: Agglomerate Pattern

input : Image I ; $pixel_i = (x_i, y_i, r_i, g_i, b_i)$; probability labels
output: Agglomerate pattern perturbation r

for $i \leftarrow 0$ **to** N_{aggl} **do**
  First operation : (j=0,2 & n=2)
  for $j \leftarrow 0$ **to** n **do**
    $pixel_i[x, j, color]$
  end
  Second operation : (j=0,2 & n=2)
  for $j \leftarrow 0$ **to** n **do**
    $pixel_i[(x + n), j, color]$
  end
  Third operation : (k=n/2)
  $pixel_i[k, k, color]$
  pixel $\leftarrow$**pixel**$_i$
end

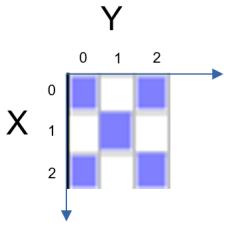


Figure 8.7: Agglomerate pattern

The **Patch Pattern** (Figure 8.8) is divided into three cases:

a. **Vertical patch**: 2 pixels are arranged sequentially along x

b. **Diagonal patch**: 2 pixels are arranged along the diagonal

c. **Line pattern**: 4 or more pixel are arranged sequentially along x

Algorithm 2: Patch Pattern

input : Image I ; $pixel_i = (x_i, y_i, r_i, g_i, b_i)$; probability labels
output: Patch perturbation r

begin
  switch K **do**
    case a **do**
      for $i \leftarrow 0$ **to** N_a **do**
        for $j \leftarrow 0$ **to** n **do**
          $pixel_i[j, y, color]$
        end
        pixel $\leftarrow$**pixel**$_i$
      end
    end
    case b **do**
      for $i \leftarrow 0$ **to** N_b **do**
        for $j \leftarrow 0$ **to** n **do**
          $pixel_i[j, j, color]$
        end
        pixel $\leftarrow$**pixel**$_i$
      end
    end
    case c **do**
      Line Pattern: it is a special case of patch pattern (case a) when $n \geq 4$
    end
  end
end

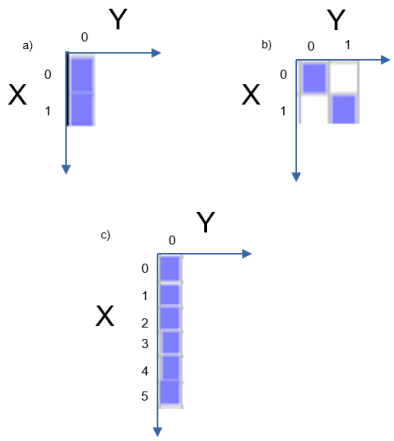


Figure 8.8: Patch pattern

In rainy conditions, the water drops are mainly concentrated in the background of the image. The creation of a **V rule** allows to simulate this effect.

Algorithm 3: Rain Attack

input : Image I ; $pixel_i = (x_i, y_i, r_i, g_i, b_i)$; probability labels, color = [208, 209, 214]
output: Rain attack perturbation r

1 Use agglomerate patterns in the first line
2 Add several patch patterns (case a, case b) and the line patterns until x/2
3 After the x/2 change pixels following the **V rule** through the agglomerate patterns

Figure 8.9: Rain Attack

The idea is to concentrate the drops by combining **agglomerate patterns** and **path patterns** following the shape of the V, as shown in Figure 8.9.

The results in Table 8.1 demonstrate that **Rain Attack** works well for **LeNet** ($ASR = 72\%$) and **ResNet** ($ASR = 67\%$), while in **CapsNet** is not so efficient ($ASR = 36\%$).

CIFAR10	**LeNet**	**ResNet**	**CapsNet**
ASR	72 %	67 %	36 %

Table 8.1: Attack Success Rate of Rain Attack

8.2.2 Snow Attack

The second attack is the **Snow Attack** (Figure 8.10): it is based on dividing the image into three parts: I,II and III. In parts I and III, the pattern is the same where there is a **hole** between pixels, while in the central part (II) there are more pixels.

Algorithm 4: Snow Attack

input : Image I ; $pixel_i = (x_i, y_i, r_i, g_i, b_i)$;
probability labels, color = [249, 242, 249]
output: Snow attack perturbation r

I and III sections:
Case a : (k=0,4,8... & j=0,3,6...)
for $i \leftarrow 0$ **to** N_{snow1} **do**
  for $k \leftarrow 0$ **to** n **do**
    for $j \leftarrow 0$ **to** m **do**
      $pixel_i[(x+k),(y+j),color]$
    end
  end
  pixel $\leftarrow$**pixel**$_i$
end
Case b : (k=2,6,10... & j=0,6,12...)
for $i \leftarrow 0$ **to** N_{snow1} **do**
  for $k \leftarrow 2$ **to** n **do**
    for $j \leftarrow 0$ **to** m **do**
      $pixel_i[(x+k),(y+j),color]$
    end
  end
  pixel $\leftarrow$**pixel**$_i$
end
II section : (k=0,3,6... & j=0,3,6...)
for $i \leftarrow 0$ **to** N_{snow2} **do**
  for $k \leftarrow 0$ **to** n **do**
    for $j \leftarrow 0$ **to** m **do**
      $pixel_i[(x+k),(y+j),color]$
    end
  end
  pixel $\leftarrow$**pixel**$_i$
end

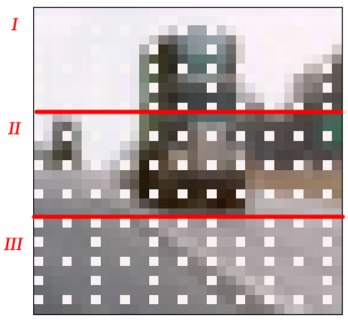


Figure 8.10: Snow Attack

The results are similar to the rain attack : **snow attack** works well for **LeNet** ($ASR = 77\%$) and **ResNet** ($ASR = 75{,}5\%$), while in **CapsNet** is not so efficient ($ASR = 28\%$), as shown in Table 8.2.

CIFAR10	**LeNet**	**ResNet**	**CapsNet**
ASR	77 %	75.5 %	28 %

Table 8.2: Attack Success Rate of Snow Attack

8.2.3 Hail Attack

The **Hail Attack** is a combination of pixels adding several **hail patterns** which are agglomerates of 8 pixels, as shown in Figure 8.11.

Algorithm 5: Hail Attack

input : Image I ; $pixel_i = (x_i, y_i, r_i, g_i, b_i)$;
probability labels, color = [249, 242, 249]
output: Hail attack perturbation r

Hail pattern:
for $i \leftarrow 0$ **to** N_{hail} **do**
  First operation : (j=0,2 & n=2)
  for $j \leftarrow 0$ **to** n **do**
    $pixel_i[x, j, color]$
  end
  Second operation : (j=1,2 & m=2)
  for $j \leftarrow 0$ **to** m **do**
    $pixel_i[(x + j), (y + 1), color]$
  end
  $pixel_i[(x + m), (y + 2), color]$
  Third operation : (j=0,2,3 & o=3)
  for $j \leftarrow 0$ **to** o **do**
    $pixel_i[(x + m + 1), j, color]$
  end
  pixel $\leftarrow$**pixel**$_i$
end
Adding several **hail patterns** as desired

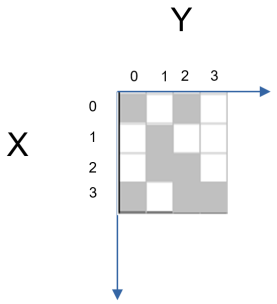


Figure 8.11: Hail Attack

In this case the **Hail Attack** allows to have the following results (Table 8.3): **LeNet** ($ASR = 82{,}5\%$) , **ResNet** ($ASR = 78{,}5\%$) and **CapsNet** ($ASR = 63\%$). It is possible to conclude that the Hail Attack performs very well in **all** models.

CIFAR10	LeNet	ResNet	CapsNet
ASR	82.5 %	78.5 %	63 %

Table 8.3: Attack Success Rate of Hail Attack

8.3 Remarks

In conclusion, taking into account the results contained in Figure 8.12, it is necessary to make two considerations:

- All types of **Pattern Attack** work well both in LeNet and ResNet models. ($ASR > 65\%$)
- **Hail Attack** is the best attack : it performs better than other attacks in all cases

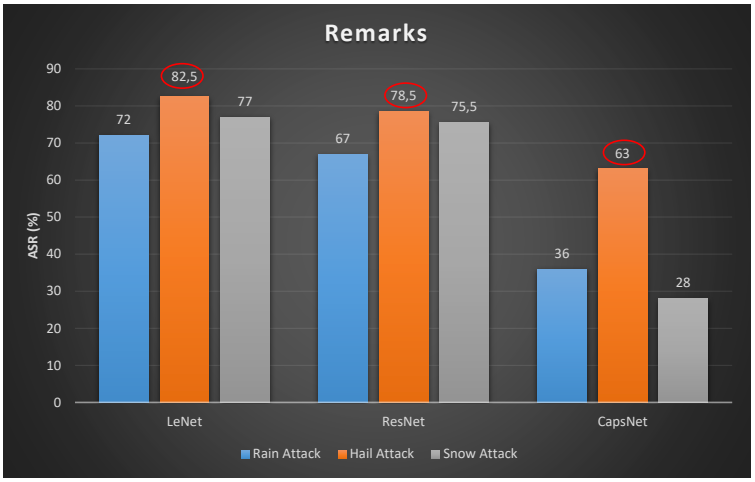


Figure 8.12: ASR of Pattern Attack

In Figure 8.13 examples of misclassification are shown for each attack: it is possible to see that the model was deceived, the predicted class is different from the real one.

Figure 8.13: Examples of misclassification for all attacks

Chapter 9

Conclusion and Future works

In recent years, adversarial attacks have become a fundamental topic in the field of Convolutional Neural Networks and in Capsule Networks. In this thesis, after evaluating the performance of CapsNets on various datasets, ample space was given to the robustness of CNNs and CapsNets against adversarial attacks on Medical Images and Remote sensing datasets. In particular, attention must be paid to medical applications, where a misclassification can mean a wrong diagnosis instead of a correct one. Later, in addition to the impact of the Vote Attack on CapsNet, a new methodology was formulated that made it possible to simulate real situations in conditions of atmospheric phenomena : the Pattern Attack has achieved excellent results.
The purpose of this work is to make an important contribution to the field of security. In Deep Learning context, CNNs and CapsNets will still be used in many tasks and it is necessary to try to improve their performances and robustness through new kinds of defense in order to find countermeasures against several adversarial attacks that are created every day.
The steps of future research may be different: for the first part of analysis, the CapsNet models will be applied to more complex datasets such as ImageNet or ObjectNet trying to obtain good classification performances. Moreover, in the context of Pattern Attack, the future developments consist of applying the attack on other higher resolution datasets and on other CapsNet models such as DeepCaps, YaoCaps and FMCaps.

Bibliography

[1] Lu Sun, Mingtian Tan, and Zhe Zhou. «A survey of practical adversarial example attacks». In: *Cybersecurity* (2018) (cit. on p. 3).

[2] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. *Adversarial Examples: Attacks and Defenses for Deep Learning*. 2018. arXiv: 1712.07107v3 [cs.LG] (cit. on p. 3).

[3] Kui Ren, Tianhang Zheng, Zhan Qin, and Xue Liu. *Adversarial Attacks and Defenses in Deep Learning*. 2020. DOI: https://doi.org/10.1016/j.eng.2019.12.012 (cit. on p. 3).

[4] Alberto Marchisio, Giorgio Nanfa, Faiq Khalid, Muhammad Abdullah Hanif, Maurizio Martina, and Muhammad Shafique. *CapsAttacks: Robust and Imperceptible Adversarial Attacks on Capsule Networks*. 2019. arXiv: 1901.09878v2 [cs.LG] (cit. on p. 3).

[5] Felix Michels, Tobias Uelwer, Eric Upschulte, and Stefan Harmeling. *On the Vulnerability of Capsule Networks to Adversarial Attacks*. 2019. arXiv: 1906.03612v1 [cs.LG] (cit. on p. 3).

[6] Yao Qin, Nicholas Frosst, Sara Sabour, Colin Raffel, Garrison Cottrell, and Geoffrey Hinton. *Detecting and Diagnosing Adversarial Images with Class-Conditional Capsule Reconstructions*. 2020. arXiv: 1907.02957v2 [cs.LG] (cit. on p. 3).

[7] Yao Qin, Nicholas Frosst, Colin Raffel, Garrison Cottrell, and Geoffrey Hinton. *Deflecting Adversarial Attacks*. 2020. arXiv: 2002.07405v1 [cs.LG] (cit. on p. 3).

[8] Antonio De Marco. «Capsule Networks Robustness against Adversarial Attacks and Affine Transformations». In: *Politecnico di Torino, Master Thesis* (2020) (cit. on p. 3).

[9] Stanley Cohen. «Chapter 2 - The basics of machine learning: strategies and techniques». In: *Artificial Intelligence and Deep Learning in Pathology*. Elsevier, 2021, pp. 13–40. DOI: https://doi.org/10.1016/B978-0-323-67538-3.00002-6 (cit. on p. 6).

[10] F. Rosenblatt. *The perceptron - A perceiving and recognizing automaton*. Tech. rep. Ithaca, New York: Cornell Aeronautical Laboratory, Jan. 1957 (cit. on p. 7).

[11] *Neural Networks Part 1: Setting up the Architecture*. URL: https://cs231n.github.io/neural-networks-1/. (accessed: 1.06.2021) (cit. on p. 7).

[12] Mohamed Elgendy. *Deep Learning for Vision Systems*. Manning Publications, 2020 (cit. on pp. 8, 10–12, 14, 19, 22).

[13] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980v9 [cs.LG] (cit. on p. 11).

[14] Francois Chollet. *Deep Learning with Python*. Manning Publications, 2018 (cit. on p. 13).

[15] *Convolutional Neural Networks (CNNs / ConvNets)*. URL: https://cs231n.github.io/convolutional-networks/. (accessed: 11.06.2021) (cit. on p. 15).

[16] *The Deep Learning(.ai) Dictionary*. URL: https://towardsdatascience.com/the-deep-learning-ai-dictionary-ade421df39e4. (accessed: 1.06.2021) (cit. on p. 15).

[17] *Geoffrey Hinton talk: What is wrong with convolutional neural nets ?* URL: https://www.youtube.com/watch?v=rTawFwUvnLE&t=130s. (accessed: 10.06.2021) (cit. on pp. 16, 21).

[18] Geoffrey Hinton, Alex Krizhevsky, and Sida Wang. «Transforming Auto-encoders». In: (2011) (cit. on pp. 16, 21, 23).

[19] Sara Sabour, Nicholas Frosst, and Geoffrey Hinton. «Dynamic Routing Between Capsules». In: (July 2017). arXiv: 1710.09829v2 (cit. on pp. 16, 21, 24, 26, 27, 42).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: (Oct. 2015). arXiv: 1512.03385v1 (cit. on p. 16).

[21] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. «Intriguing properties of neural networks». In: (19 Feb 2014). arXiv: 1312.6199v4 (cit. on p. 17).

[22] Claudio Gambella, Bissan Ghaddar, and Joe Naoum-Sawaya. «Optimization problems for machine learning: A survey». In: (Sept. 2020). DOI: https://doi.org/10.1016/j.ejor.2020.08.045 (cit. on p. 17).

[23] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015 (cit. on p. 19).

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Nov 2016 (cit. on p. 19).

[25] Geoffrey E. Hinton, S. Sabour, and Nicholas Frosst. «Matrix capsules with EM routing». In: *ICLR*. 2018 (cit. on pp. 21, 24, 30).

[26] *Understanding Hinton's Capsule Networks*. URL: https://medium.com/ai%C2%B3-theory-practice-business/understanding-hintons-capsule-networks-part-i-intuition-b4b559d1159b. (accessed: 9.06.2021) (cit. on pp. 22, 23).

[27] *Sara Sabour : Introduction to Capsules*. URL: https://youtu.be/zRg3IuxaJ6I. (accessed: 8.06.2021) (cit. on pp. 23, 24).

[28] *Aurélien Géron: Capsule Networks (CapsNets) – Tutorial*. URL: https://youtu.be/pPN8d0E3900. (accessed: 7.06.2021) (cit. on p. 24).

[29] *THE MNIST DATABASE of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/. (accessed: 7.06.2021) (cit. on pp. 24, 26, 34).

[30] Jathushan Rajasegaran, Vinoj Jayasundara, Sandaru Jayasekara, Hirunima Jayasekara, Suranga Seneviratne, and Ranga Rodrigo. «DeepCaps: Going Deeper with Capsule Networks». In: (21 Apr 2019). arXiv: 1904.09546v1 (cit. on pp. 27–29).

[31] *The CIFAR10-100 dataset*. URL: https://www.cs.toronto.edu/~kriz/cifar.html. (accessed: 7.06.2021) (cit. on pp. 27, 38, 46).

[32] *The Street View House Numbers (SVHN) Dataset*. URL: http://ufldl.stanford.edu/housenumbers/. (accessed: 7.06.2021) (cit. on pp. 27, 35).

[33] Han Xiao, Kashif Rasul, and Roland Vollgraf. «Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms». In: *CoRR* abs/1708.07747 (2017). arXiv: 1708.07747. URL: http://arxiv.org/abs/1708.07747 (cit. on p. 27).

[34] Yao-Hung Hubert Tsai, Nitish Srivastava, Hanlin Goh, and Ruslan Salakhutdinov. *Capsules with Inverted Dot-Product Attention Routing*. 2020. arXiv: 2002.04764v2 [cs.LG] (cit. on pp. 30, 31).

[35] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450v1 [stat.ML] (cit. on p. 30).

[36] Lei Zhao, Xiaohui Wang, and Lei Huang. *An Efficient Agreement Mechanism in CapsNets By Pairwise Product*. 2020. arXiv: 2004.00272v1 [cs.CV] (cit. on pp. 31, 32).

[37] Steffen Rendle. «Factorization Machines». In: *2010 IEEE International Conference on Data Mining*. 2010. DOI: 10.1109/ICDM.2010.127 (cit. on p. 31).

[38] Karim Ahmed and L. Torresani. «STAR-Caps: Capsule Networks with Straight-Through Attentive Routing». In: *NeurIPS*. 2019 (cit. on p. 32).

[39] Aryan Mobiny, Pietro Antonio Cicalese, and Hien Van Nguyen. *Trans-Caps: Transformer Capsule Networks with Self-attention Routing*. 2021. URL: https://openreview.net/forum?id=BUPIRa1D2J (cit. on p. 32).

[40] Fabio De Sousa Ribeiro, Georgios Leontidis, and Stefanos D. Kollias. «Capsule Routing via Variational Bayes». In: *CoRR* abs/1905.11455 (2019). arXiv: 1905.11455. URL: http://arxiv.org/abs/1905.11455 (cit. on p. 32).

[41] Vittorio Mazzia, Francesco Salvetti, and Marcello Chiaberge. «Efficient-CapsNet: Capsule Network with Self-Attention Routing». In: *CoRR* abs/2101.12491 (2021). arXiv: 2101.12491. URL: https://arxiv.org/abs/2101.12491 (cit. on p. 32).

[42] *Keras Documentation*. URL: https://keras.io/api/. (accessed: 04.05.2021) (cit. on p. 34).

[43] *TensorFlow Documentation*. URL: https://www.tensorflow.org/api_docs. (accessed: 06.05.2021) (cit. on p. 34).

[44] *PyTorch Documentation*. URL: https://pytorch.org/docs/stable/index.html. (accessed: 07.05.2021) (cit. on p. 34).

[45] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. «Reading Digits in Natural Images with Unsupervised Feature Learning». In: (2011) (cit. on p. 34).

[46] Alex Krizhevsky. «Learning Multiple Layers of Features from Tiny Images». In: (2009) (cit. on pp. 37, 42).

[47] Jindong Gu, Baoyuan Wu, and Volker Tresp. «Effective and Efficient Vote Attack on Capsule Networks». In: *International Conference on Learning Representations*. arXiv: 2102.10055v1 (cit. on pp. 42, 68, 81).

[48] Luke N. Darlow, Elliot J. Crowley, Antreas Antoniou, and Amos J. Storkey. «CINIC-10 is not ImageNet or CIFAR-10». In: (Feb. 2018). arXiv: 1810.03505v1 (cit. on p. 46).

[49] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. «A Downsampled variant of IMAGENET as alternative to the CIFAR Datasets». In: (23 Aug 2017). arXiv: 1707.08819v3 (cit. on p. 46).

[50] *The CINIC-10 dataset*. URL: https://www.kaggle.com/mengcius/cinic10. (accessed: 7.06.2021) (cit. on p. 46).

[51] Xiaoman Qi et al. «MLRSNet: A Multi-label High Spatial Resolution Remote Sensing Dataset for Semantic Scene Understanding». In: (Oct. 2020). arXiv: 2010.00243v1 (cit. on pp. 51, 52).

[52] Xiao Qi, Lloyd Brown, David Foran, John Nosher, and Ilker Hacihaliloglu. «Chest X-ray image phase features for improved diagnosis of COVID-19 using convolutional neural network». In: (Jan. 2021). DOI: https://doi.org/10.1007/s11548-020-02305-w (cit. on pp. 51, 54, 55).

[53] *MLRS Dataset*. URL: https://data.mendeley.com/datasets/7j9bv9vwsx/2. (accessed: 03.05.2021) (cit. on p. 51).

[54] *Coronavirus*. URL: https://www.who.int/health-topics/coronavirus#tab=tab_1. (accessed: 5.05.2021) (cit. on p. 54).

[55] *COVID-Ti Dataset: the largest COVID-19 Dataset (CXR and Enhanced CXR)*. URL: https://www.kaggle.com/endiqq/largest-covid19-dataset?select=CXR. (accessed: 20.03.2021) (cit. on pp. 54, 55).

[56] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2015. arXiv: 1412.6572v3 [stat.ML] (cit. on pp. 64, 65).

[57] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. «Adversarial examples in the physical world». In: *CoRR* (2017). arXiv: 1607.02533v4 (cit. on pp. 65, 85).

[58] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jianguo Li, Xiaolin Hu, and Jun Zhu. «Boosting Adversarial Attacks with Momentum». In: *CoRR* (2018). arXiv: 1710.06081v3 (cit. on p. 65).

[59] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. *Towards Deep Learning Models Resistant to Adversarial Attacks*. 2019. arXiv: 1706.06083v4 [stat.ML] (cit. on p. 66).

[60] Nicolas Papernot et al. *Technical Report on the CleverHans v2.1.0 Adversarial Examples Library*. 2018. arXiv: 1610.00768v6 [cs.LG] (cit. on p. 66).

[61] Jonas Rauber, Wieland Brendel, and Matthias Bethge. *Foolbox: A Python toolbox to benchmark the robustness of machine learning models*. 2018. arXiv: 1707.04131v3 [cs.LG] (cit. on p. 66).

[62] Gavin Weiguang Ding, Luyu Wang, and Xiaomeng Jin. *advertorch v0.1: An Adversarial Robustness Toolbox based on PyTorch*. 2019. arXiv: 1902.07623v1 [cs.LG] (cit. on p. 66).

[63] *CleverHans Documentation*. URL: https://cleverhans-nottombrown-fork.readthedocs.io/en/latest/. (accessed: 03.05.2021) (cit. on p. 66).

[64] *FoolBox Documentation*. URL: https://foolbox.readthedocs.io/en/stable/. (accessed: 03.05.2021) (cit. on p. 66).

[65] Anish Athalye, Nicholas Carlini, and David Wagner. *Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples*. 2018. arXiv: 1802.00420v4 [cs.LG] (cit. on p. 66).

[66] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. *On Evaluating Adversarial Robustness*. 2019. arXiv: 1902.06705v2 [cs.LG] (cit. on p. 66).

[67] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. *DeepFool: a simple and accurate method to fool deep neural networks*. 2016. arXiv: 1511.04599v3 [cs.LG] (cit. on p. 68).

[68] David Mickisch, Felix Assion, Florens Greßner, Wiebke Günther, and Mariele Motta. *Understanding the Decision Boundary of Deep Neural Networks: An Empirical Study*. 2020. arXiv: 2002.01810 [cs.LG] (cit. on p. 68).

[69] Nicholas Carlini and David Wagner. *Towards Evaluating the Robustness of Neural Networks*. 2017. arXiv: 1608.04644v2 [cs.CR] (cit. on p. 68).

[70] Jérôme Rony, Luiz G. Hafemann, Luiz S. Oliveira, Ismail Ben Ayed, Robert Sabourin, and Eric Granger. *Decoupling Direction and Norm for Efficient Gradient-Based L2 Adversarial Attacks and Defenses*. 2019. arXiv: 1811.09600v3 [cs.CV] (cit. on p. 68).

[71] Xingjun Ma, Yuhao Niu, Lin Gu, Yisen Wang, Yitian Zhao, James Bailey, and Feng Lu. *Understanding Adversarial Attacks on Deep Learning Based Medical Image Analysis Systems*. 2020. arXiv: 1907.10456v2 [cs.CV] (cit. on p. 69).

[72] Nina Narodytska and Shiva Prasad Kasiviswanathan. *Simple Black-Box Adversarial Perturbations for Deep Networks*. 2016. arXiv: 1612.06299v1 [cs.LG] (cit. on p. 85).

[73] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. «One Pixel Attack for Fooling Deep Neural Networks». In: (Oct. 2019). arXiv: 1710.08864v7 [cs.LG] (cit. on pp. 85, 87).

[74] Diego Gragnaniello, Francesco Marra, Giovanni Poggi, and Luisa Verdoliva. *Perceptual Quality-preserving Black-Box Attack against Deep Learning Image Classifiers*. 2020. arXiv: 1902.07776v3 [cs.CV] (cit. on p. 85).

[75] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. *Adversarial Patch*. 2018. arXiv: 1712.09665v2 [cs.CV] (cit. on pp. 85, 86).

[76] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. *Robust Physical-World Attacks on Deep Learning Visual Classification*. 2018. arXiv: 1707.08945v5 [cs.CV] (cit. on p. 86).

[77] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. «Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition». In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. Oct. 2016 (cit. on p. 86).

[78] Liming Zhai, Felix Juefei-Xu, Qing Guo, Xiaofei Xie, Lei Ma, Wei Feng, Shengchao Qin, and Yang Liu. «It's Raining Cats or Dogs? Adversarial Rain Attack on DNN Perception». In: (Sept. 2020). arXiv: 2009.09205v1 [cs.CV] (cit. on p. 86).

[79] *Water-Drop: come fotografare una goccia d'acqua*. URL: https://www.inabottle.it/it/cultura/water-drop-tecnica-fotografare-goccia-acqua. (accessed: 06.06.2021) (cit. on p. 87).

[80] *Snowflakes and avalanches*. URL: https://www.sciencenewsforstudents.org/article/snowflakes-and-avalanches. (accessed: 06.06.2021) (cit. on p. 87).

[81] *Aggiornamento Meteo GP Stiria – Arriva il diluvio, qualifiche realmente a rischio*. URL: https://f1ingenerale.com/f1-aggiornamento-meteo-gp-stiria-arriva-il-diluvio-qualifiche-realmente-a-rischio/. (accessed: 08.06.2021) (cit. on p. 90).

[82] *Maltempo: divieto di balneazione da Torrette al Passetto*. URL: https://www.meteoweb.eu/2019/08/maltempo-divieto-di-balneazione-da-torrette-al-passetto/1301567/. (accessed: 07.06.2021) (cit. on p. 90).

[83] *Meteo, Pioggia in arrivo: le previsioni per martedì 2 ottobre*. URL: https://www.sardegnalive.net/news/in-sardegna/27621/meteo-pioggia-in-arrivo-le-previsioni-per-martedi-2-ottobre. (accessed: 09.06.2021) (cit. on p. 90).