UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ

CENTRO DE INVESTIGACIÓN Y ESTUDIOS DE POSGRADO

FACULTAD DE INGENIERÍA

Computational Model Based on Capsule Networks for Image Classification

TESIS PROFESIONAL

PARA OBTENER EL GRADO DE:

DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN

Presenta:

M.I.E. GABRIELA RANGEL RAMÍREZ

Asesor de tesis:

Dr. JUAN CARLOS CUEVAS TELLO

San Luis Potosí, S.L.P., Enero de 2024

UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ

CENTRO DE INVESTIGACIÓN Y ESTUDIOS DE POSGRADO

FACULTAD DE INGENIERÍA

Modelo Computacional Basado en Redes de Cápsula para Clasificación de Imágenes

TESIS PROFESIONAL

PARA OBTENER EL GRADO DE:

DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN

Presenta:

M.I.E. GABRIELA RANGEL RAMÍREZ

Asesor de tesis:

Dr. JUAN CARLOS CUEVAS TELLO

San Luis Potosí, S.L.P., Enero de 2024

UASLP
Universidad Autónoma de San Luis Potosí

FACULTAD DE INGENIERÍA

17 de agosto 2023

M. I. GABRIELA RANGEL RAMÍREZ
P R E S E N T E.

En atención a su solicitud de Temario, presentada por el **Dr. Juan Carlos Cuevas Tello** Asesor de la Tesis que desarrollará Usted, con el objeto de obtener el Grado de **Doctora en Ciencias de la Computación**, me es grato comunicarle que en la sesión del H. Consejo Técnico Consultivo celebrada el día 17 de agosto del presente año, fue aprobado el Temario propuesto:

TEMARIO:

"Modelo computacional basado en redes de cápsulas para clasificación de imágenes"

1. Introducción.
2. Marco teórico de tareas de visión computacional.
3. Redes neuronales convolucionales y sus limitaciones.
4. Redes de cápsulas como una solución alternativa.
5. Experimentos y resultados.
6. Conclusiones y trabajo futuro.
   Referencias.
   Apéndice.

"MODOS ET CUNCTARUM RERUM MENSURAS AUDEBO"

A T E N T A M E N T E

DR. EMILIO JORGE GONZÁLEZ GALVÁN
DIRECTOR.

UNIVERSIDAD AUTONOMA DE SAN LUIS POTOSI
FACULTAD DE INGENIERIA
DIRECCION

Copia. Archivo.
*etn.

ANIVERSARIO DE LA AUTONOMÍA UASLP 2023

www.uaslp.mx

Av. Manuel Nava 8
Zona Universitaria • CP 78290
San Luis Potosí, S.L.P.
tel. (444) 826 2330 al39
fax (444) 826 2336

Agradecimientos

A mis padres, hermanos, familia y polas quienes siempre creyeron en mi más que yo misma.

A mi asesor el Dr. Juan Carlos Cuevas Tello quien confió en mi desde el principio y siempre me mostró su apoyo y su buena disposición.

Al Dr. Mariano Rivera, que siempre ofreció disponibilidad , comentarios y retroalimentación muy asertiva para la realización de este proyecto.

A los doctores Cesar Augusto Puente Montejano y Jose Ignacio Núñez Varela, miembros de mi comité de tesis quienes a lo largo de los avances me acompañaron en este proceso con mucha paciencia.

A mis amigos de toda la vida que siempre estuvieron ahí para apoyarme.

A mis compañeros de posgrado que me dieron valiosas lecciones e hicieron el camino más llevadero.

A todas las personas que sin saberlo contribuyeron a la terminación de este trabajo, dándome apoyo, comida o una buena plática.

Al TecSuperior por darme las facilidades para realizar este posgrado.

Al Conacyt por la beca otorgada.

Resumen

La tarea de clasificación de imágenes consiste en asignar una clase definida a una imagen. Esta tarea es considerada sencilla para los humanos, sin embargo, es una tarea compleja para las computadoras debido a todos los procedimientos que se tiene que realizar para que la computadora adquiera esta habilidad. Debido a las múltiples aplicaciones de esta tarea, se han propuesto una amplia variedad de modelos computacionales con diferentes enfoques. Uno de estos enfoques son las Redes Neuronales Artificiales, las cuales están inspiradas en la biología del cerebro humano y utilizan muchas herramientas estadísticas para lograr el objetivo de clasificación de imágenes. En la última década, los modelos basados en Redes Neuronales Artificiales han tomado relevancia por sus resultados obtenidos, sobrepasando incluso el error humano en ciertas tareas de clasificación. A pesar de sus buenos resultados en imágenes seleccionadas, estos algoritmos sufren cuando usan como entradas imágenes reales, debido a que la forma en la que adquieren sus habilidades aun no es la más optima. Por estas razones nuevos enfoques se siguen desarrollando como las Redes de Cápsula, las cuales consisten en extraer la información de las imágenes en forma vectorial y encapsularla para no perder la información que ha surgido. Debido al aumento de la complejidad en el manejo de la información en estas redes únicamente ha sido probado con imágenes sencillas y de tamaño pequeño. Por lo que esta tesis propone un nuevo modelo computacional basado en Redes de Cápsula que combina lo mejor de los algoritmos propuestos hasta el momento para el manejo de imágenes reales y de mayor tamaño, como lo son las Redes Neuronales Convolucionales. Como resultado, se realizan tres contribuciones relevantes al estado del arte: *i)* el estudio y la agrupación de las limitaciones encontradas en los algoritmos considerados como el estado del arte en la tarea de clasificación de imágenes, *ii)* la descripción de todos los elementos a considerar para la implementación de un modelo computacional desde cero, *iii)* el diseño de un modelo computacional basado en la combinación de dos enfoques capaz de clasificar imágenes complejas. Los resultados experimentales muestran que la implementación del modelo produce resultados comparables con algoritmos actuales sin necesidad de utilizar técnicas especiales para los datos de entrada.

Abstract

The image classification task consists of assigning a defined class to an image. This task is considered simple for humans, however, it is a complex task for computers due to all the procedures that have to be performed for the computer to acquire this skill. Due to the multiple applications of this task, a wide variety of computational models with different approaches have been proposed. The Artificial Neural Networks are one of this approaches, which are inspired by the biology of the human brain and use many statistical tools to achieve the goal of image classification. In the last decade, the models based on Artificial Neural Networks has gained relevance for its results, surpassing even human error in certain classification tasks. Despite their good results on selected images, these algorithms suffer when using real images as inputs, because the way they acquire their skills is not yet optimal. For these reasons new approaches are still being developed such as Capsule Networks, which extract the information from the images in vector form and encapsulating it in order not to lose it. Due to the increasing complexity of information handling these new algorithms have only been tested with simple and small size images. Therefore, this thesis proposes a new computational model based on Capsule Networks which combines the best of the algorithms proposed so far for handling real and larger images like Convolutional Neural Networks. As a result, three relevant contributions to the state of the art are made: *i)* the study and clustering of the limitations found in the algorithms considered as the state of the art in the task of image classification, *ii)* the description of all the elements to consider for the implementation of a computational model from scratch, *iii)* the design of a computational model based on the combination of two approaches capable of classifying complex images. Experimental results show that the implementation of the model produces results comparable with current algorithms without the need to use special techniques for the input data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Intelligence (AI) can be described as the effort to automate intellectual tasks normally performed by humans [4]. An important branch of AI is computer vision. Computer vision is an interdisciplinary field that deals with how computers can acquire high-level understanding from digital images or videos [10]. This field has been extensively studied from different perspectives, such as robotics, mathematics, statistics, and computer science. From a computer science perspective, the goal of computer vision is to interpret and understand the visual world, i.e., to identify people, name objects, or infer the geometry of things. Consequently, computer vision involves a variety of tasks, including classifying a set of images into a predefined set of categories, detecting and localizing objects into bounding boxes, and segmenting an image into smaller components or sub-objects, among others.

Machine Learning (ML) is a branch of Artificial Intelligence (AI) where programs are taught to identify patterns in data and subsequently make decisions and predictions from these patterns. Machine Learning is defined as a field of study that gives computers the ability to learn without being explicitly programmed [3, 11]. Also, ML can be classified according to the type of supervision that is used during training. This thesis focuses on the category of supervised learning. In addition, the ML field uses a data-driven approach. Unlike classic programming, where the data and the rules are the inputs for obtaining the desired output. Now the algorithm is fed with data

and the desired output, where the challenge is to perform the reasoning that leads to the desired result. This ability is obtained through a feature extraction stage performed by either a human or an algorithm, and then the classification algorithm as shown on top of the Figure 1.1.

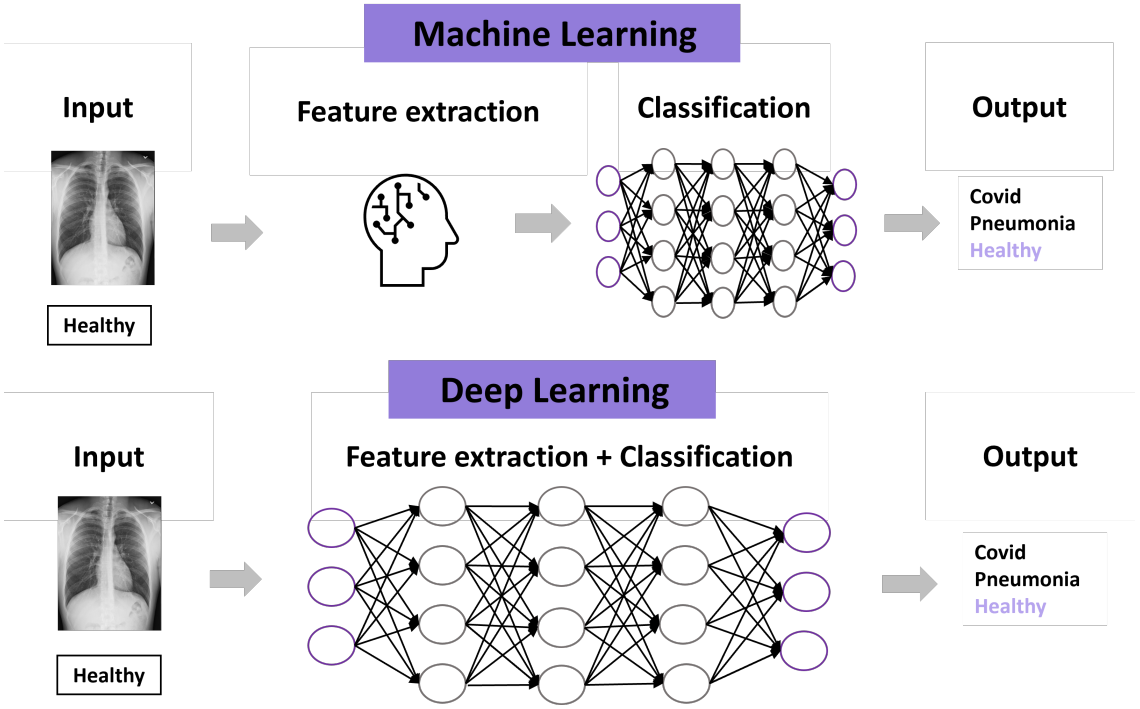


Figure 1.1. Differences between Deep Learning and Machine Learning.

A wide range of Machine Learning models have been proposed but there is one branch of ML that excels in solving computer vision tasks, called Artificial Neural Networks (ANNs) architecture. These networks are inspired by networks of biological neurons found in the human brain [3]. ANNs are versatile, powerful, and scalable, making them ideal for tackling large and highly complex ML tasks. Finally, in the last decade, a new approach have exhibited superior performance compared to other ML algorithms in the image classification task [12]. This superior performance is possible due to three important factors: a great amount of data available, powerful hardware components, and new architectures [13]. This approach is called Deep Learning (DL) which uses ANNs as a backbone to process data through various layers of algorithms and reach an accurate decision without human intervention, where the feature extraction and the classification task is performed by the ANNs at the same time, see at the bottom in Figure 1.1.

Deep Learning architectures have been considered a revolution in the field of computer vision, especially the Convolutional Neural Networks (CNNs) architecture. CNNs includes feature extraction to analyze image inputs, without a preprocessing stage. The CNNs popularity is thanks to the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [12], which is one of the most important challenges around the world in computer vision tasks, it started in 2010, known simply as ImageNet. Since 2012, the CNNs won the ImageNet competition. From ImageNet, several successful Deep Learning architectures have emerged including: AlexNet [14], ZFNet [15], GoogLeNet [16], and ResNet [17], to mention just a few. Since the appearance of these architectures, there has been an exponential increase in the use of CNNs in scientific papers [18].

Due to these satisfactory results, there is a great diversity of applications of these networks in many fields. For example, the diagnosis of plant diseases to save crops [19], the detection of endangered species in complex environments [20], the prediction of the 3D structure of the protein from its amino acid sequence [21], the monitoring of water quality through the implementation of biomonitoring using macroinvertebrates as indicators [22], facial recognition for security systems and industrial applications, among many others [23]. Furthermore, there are several publications that use CNN for medical diagnostics; *e.g.*, pulmonary diseases as COVID or pneumonia [24], detection of lung cancer [25], brain tumors [26, 27], and Alzheimer's disease [28]. In particular, some articles report on a procedure for the detection of breast cancer based on computerized tomography scans [29, 30, 31] and also report on a method for identifying genetic disorders by analyzing facial gestures [32].

However, several researchers have started to report that CNN models are brittle and that their performance can be severely degraded in real-world applications [33, 34, 7, 35, 36]. When CNNs cannot achieve an acceptable classification result, we say that CNNs have limited performance. Several papers have identified some of these CNN limitations, but most of them report only one type of limitation at most. So, one of the contributions of this thesis is to provide a description and analysis of all current CNN limitations, and we propose to group them into the following categories: data labeling, spatial relationship, invariance transformation, and adversarial attacks. It is important to note that each limitation is still an open area of research.

Due to all these limitations, it is necessary to look for other approaches that allow us to improve these algorithms. A novel approach that addresses all of these limitations is called Capsule Networks (CapsNets). These networks were introduced by Sabourn *et al.* in 2017 [7], to improve the way the network passes information through its layers [8, 37]. For example, Figure 1.2 shows the conceptual differences between CNNs and CapsNet. CNNs excel at finding patterns in images and detecting the components of the images. However, CNNs do not care about the spatial relationship of the components, i.e. when detecting two eyes, a nose and a mouth, the network infers that it is a face even if its location is not correct. On the other hand, CapsNets besides detecting the components of the image make a prediction taking into account the location of its component and the location of the other predictions to find coincidences that allow building a coherent image. In order to achieve this behavior, it is necessary to encode a large amount of information. These networks analyze images by switching from scalar operations to vector operations. This analysis allows us to obtain good classification results with few training images. In addition to classifying correctly, the CapsNet approach includes a stage in its architecture that is responsible for reconstructing the input image. In this way, the algorithm verifies that the values obtained in its training learn the most crucial image patterns.

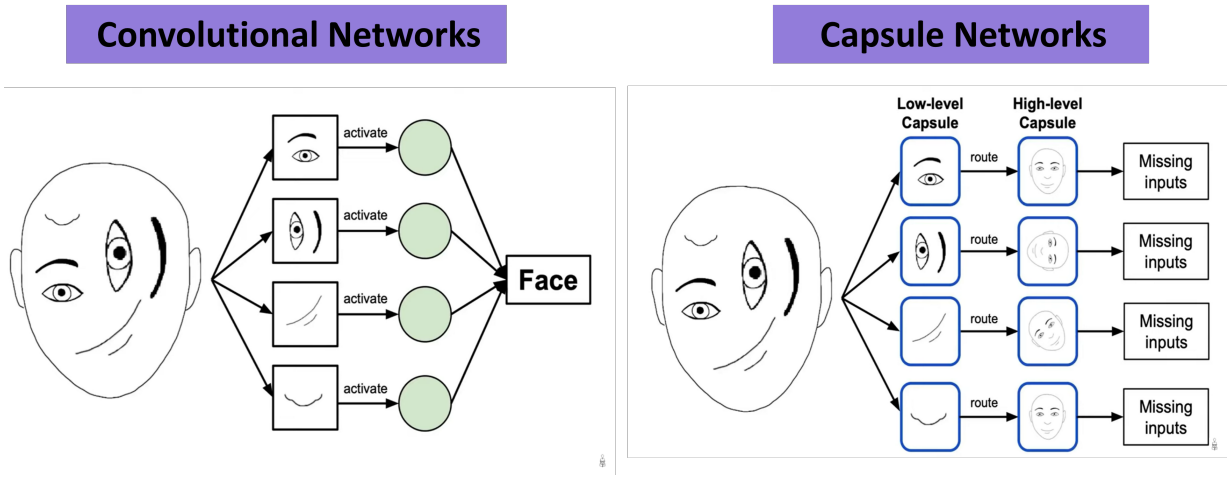


Figure 1.2. Differences between CNN and CapsNets.

According to the literature, the promising CapsNets results mentioned previously were obtained with simple image datasets such as MNIST, which is the most used dataset as benchmark for a new architecture [38]. This dataset handles small images with a size of 28 $\times$ 28 pixels.

However, CapsNets are limited when analyzing bigger size images. The architecture struggles to understand the entire context of the image, generating a large number of parameters, resulting in substantial computational effort [39]. For these reasons, some researchers have focused on combining the advantages of CNN with those of CapsNets [39], [40], [41], [42], [39], [43]. Therefore, there is still work to be done to achieve state-of-the-art results in image classification tasks with complex datasets.

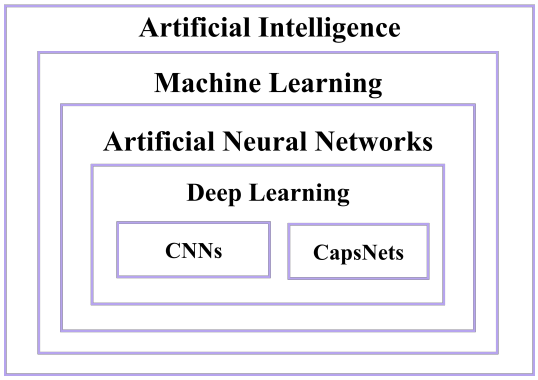


Figure 1.3. Scope of this thesis.

So, this thesis focused on the field of Deep Learning as shown in Figure 1.3. This research presents the design of a computational model called DRCaps for image classification using a medical dataset for its validation. The model is based on the combination of CapsNet and CNN architectures as shown in Figure 1.4. In addition, the DRCaps model is designed to be capable of handling complex images and to use it in a real-world problem. The DRCaps model obtains an accuracy in the image classification task of 90%.

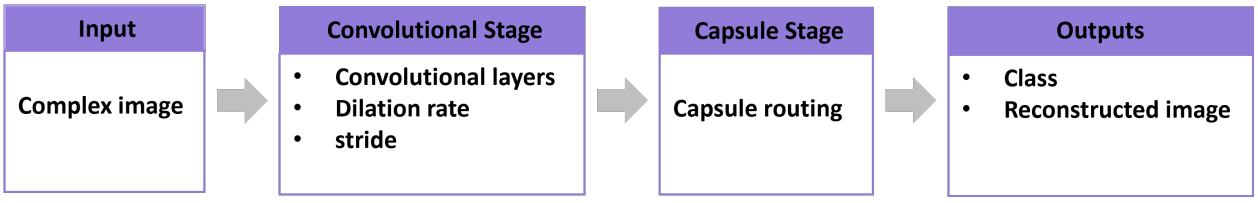


Figure 1.4. DRCaps model.

1.1 Research Question

How accurate is a computational model for image classification based on the novel CapsNets approach compared to state-of-the-art Deep Learning models?

1.2 Research Objective

1.2.1 General Objective

To develop a robust computational model based on CapsNet that can be applied to image classification tasks.

1.2.2 Specific Objectives

1. To implement the computational model of the original CapsNets architecture.

2. To validate the original CapsNet architecture with the MNIST benchmark dataset.

3. To propose improvements to the CapsNets architecture using the advantages of CNNs for handling a complex dataset.

4. To select and adjust a medical image dataset for the image classification task.

5. To build a computational model based on CapsNet architectures for the selected medical dataset.

6. To validate the computational model with the medical dataset.

1.3 Research Contributions

According to the research objective described above, the following is the list of contributions to this thesis.

- The study of CNN limitations, which are grouped into four categories: data labeling, translation invariance, adversarial examples, and spatial relationship.
- A new computational model based on the Capsule Network approach that can handle complex medical images with the use of dilated convolutions and stride hyperparameters.
- The implementation of Capsule Networks from scratch using Python, TensorFlow and Keras.

1.4 Research Methodology

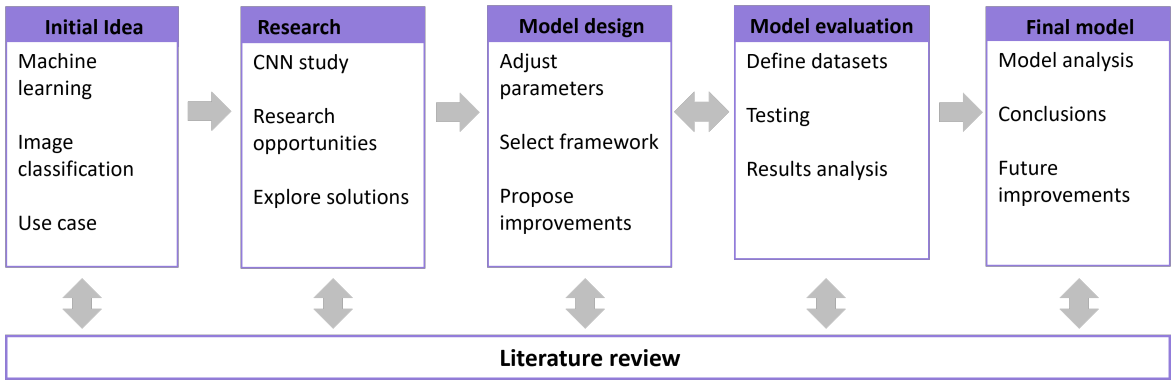


Figure 1.5. Research methodology used in this project.

This research follows the methodology shown in Figure 1.5 to achieve its objectives. Initially, a Machine Learning model was chosen to solve an image classification problem, and then applied to a real use case. To do this, a deep investigation of CNN models was conducted, as they are the state-of-the-art for image classification. This research revealed potential research opportunities and explored solutions such as the CaspNet approach. After selecting the approach, an iterative process was started between model design and model evaluation. Finally, the final configuration

of the model was obtained. All these steps were developed together with the literature review. This document presents the process of following the methodology described above, describing the findings, the difficulties found, and the proposals made during the completion of this research.

1.5 Publications

This is the list of publications generated from this thesis:

- Rangel-Ramirez, G. and Cuevas-Tello, J.C. (2018) Computational models for visual inspection in the automotive industry, XII Taller-Escuela de Procesamiento de Imagenes (PI18), CIMAT, Guanajuato, Mexico

- Rangel-Ramirez G., Cuevas-Tello J.C. (2019) Algoritmos de Aprendizaje Profundo, CONGRESO NACIONAL DE CIRCUITOS Y SISTEMAS 2019. pp. 54-55, ISBN: 978-607-535-119-3

- Rangel, G., Cuevas-Tello, J. C., Rivera, M., & Renteria, O. (2023). A Deep Learning Model Based on Capsule Networks for COVID Diagnostics through X-ray Images. Diagnostics, 13(17), 2858. https://doi.org/10.3390/diagnostics13172858 ISSN: 2075-4418

- Rangel, G., Cuevas-Tello, J.C., Nunez-Varela, J.I., Puente, C., Silva-Trujillo, A.G. (2023) A Survey on Convolutional Neural Networks and Their Performance Limitations in Image Recognition Tasks, Journal of Sensors, under review

Also, I contributed in the following publications during my Ph.D.:

- Rojas-Aranda J.L., Nunez-Varela J.I., Cuevas-Tello J.C., Rangel-Ramirez G. (2020) Fruit Classification for Retail Stores Using Deep Learning. In: Figueroa Mora K., Anzurez Marín J., Cerda J., Carrasco-Ochoa J., Martínez-Trinidad J., Olvera-López J. (eds) Pattern Recognition. MCPR 2020. Lecture Notes in Computer Science, vol 12088, pp 3-13.

Springer, Cham, DOI: 10.1007/978-3-030-49076-8_1 ISBN: 978-3-030-49075-1, eISBN: 978-3-030-49076-8

- Soriano-Mendez, M.A., Cuevas-Tello, J.C., Rangel-Ramirez G. (2019) Visual Recognition With Machine Learning Using Cloud Services, Congreso Internacional de Supercómputo (ISUM 2019)

1.6 Thesis Outline

Chapter 2: This chapter introduces the definition of computer vision from the viewpoint of this thesis. Then it examines the challenges of the image classification task, followed by a presentation of the datasets used in the experiments. Subsequently, a basic introduction to Artificial Neural Networks is provided, and the difficulties associated with their training for the image classification task are presented.

Chapter 3: This chapter explains all the conceptual foundations of CNNs. It mentions the main datasets for CNN training and presents the two datasets used in this thesis. It also describes the main CNN architectures and hyperparameters. Finally, it groups the limitations identified in the state-of-the-art CNNs into four categories.

Chapter 4: This chapter presents the novel Capsule Network approach. First, it explains how this approach works and why this new approach is important. Subsequently, the chapter makes an in-depth analysis of this architecture, mentioning its main advantages and limitations. Finally, the DRCapsnet model design process is described in detail.

Chapter 5: This chapter explains the results of the DRCaps computational model in a medical image classification task. First, it describes the experimental platform used and explains all the preprocessing needed for the medical dataset. Then, it explains the experimental setup and the result related to the DRCapsNet model.

Chapter 6: This chapter summarizes the main findings obtained from the analysis of the CapsNet approach. Finally, the chapter addresses the strengths and limitations of the CapsNet model and

proposes areas for future research.

Chapter 2

Theoretical Framework of Computer Vision Tasks

This chapter introduces the definition of computer vision. Next, it examines the challenges of the image classification task, followed by a presentation of the datasets used in the experiments. Subsequently, a basic introduction to Artificial Neural Networks is provided, and the difficulties associated with training them for the image classification task are presented.

2.1 Computer Vision

Today, computer vision is one of the most important and fastest growing research area, because computer vision is an interdisciplinary field that touches many different areas such as physics, biology, psychology, computer science, mathematics, engineering among others. It can be said that computer vision is the study of visual data [1]. In this thesis, we study computer vision from the perspective of artificial intelligence that aims to train computers to mimic the human brain to interpret and understand the visual world, that is, to identify people, name objects, or infer the geometry of things, among other tasks. The most popular computer vision tasks are:

- **Image classification**: where algorithms produce a list of object categories present in the image [12].

- **Single-Object localization**: where algorithms produce a list of objects categories present in the image, along with an axis-aligned bounding box indicating the position and scale of *one* instance of each object category [12].

- **Object detection**: where algorithms produce bounding boxes indicating the position and scale of *all* instances of *all* target object categories [12].

As mentioned above, computer vision involves many tasks, but this thesis focused only on the image classification task. Therefore, the next section describes the image classification task and the challenges it presents.

2.2 The Image Classification Task

The image classification task consists of an input image being assigned a category from a predetermined set of categories by some algorithm. As illustrated in Figure 2.1, where the cat category achieves the highest value, compared with dog, deer and bird categories. This task appears to be straightforward for humans, but it is a complex challenge for a computer. What a computer perceives in an image is different from what a human sees. Instead of a picture, the computer sees a large grid of numbers with values ranging from 0 to 255 (gray scale) in each pixel, as illustrated in Figure 2.2. Each number corresponds to one pixel of the image.

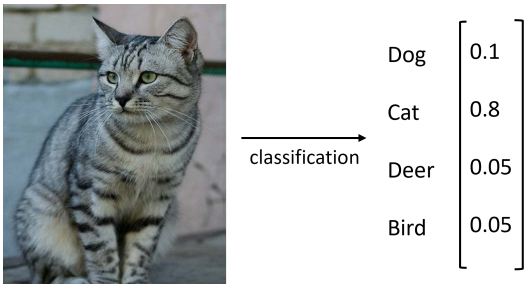


Figure 2.1. Image classification problem in computer vision [1].

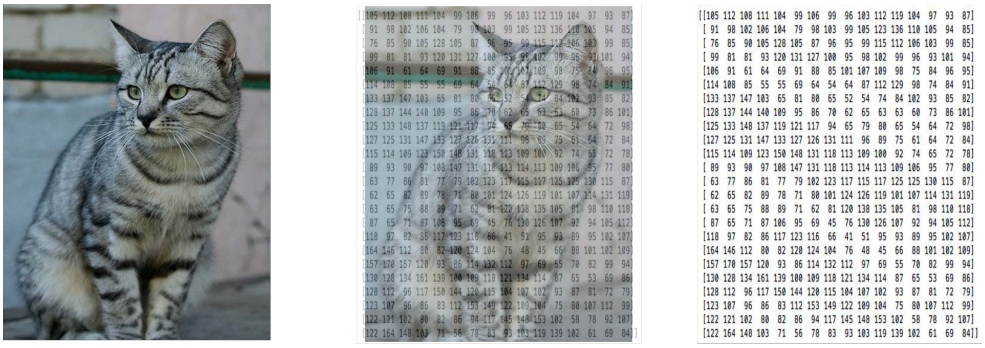


Figure 2.2. Pixel representation of an image adapted from [1].

Image classification is a very difficult problem because a small change in the image in a subtle way will cause the pixel grid to change internally. These changes could be if we perform a viewpoint variation, such as rotating the image, scaling it, or mirroring, as shown in Figure 2.3. However, it is still the same cat, so the algorithm must be robust to these changes.

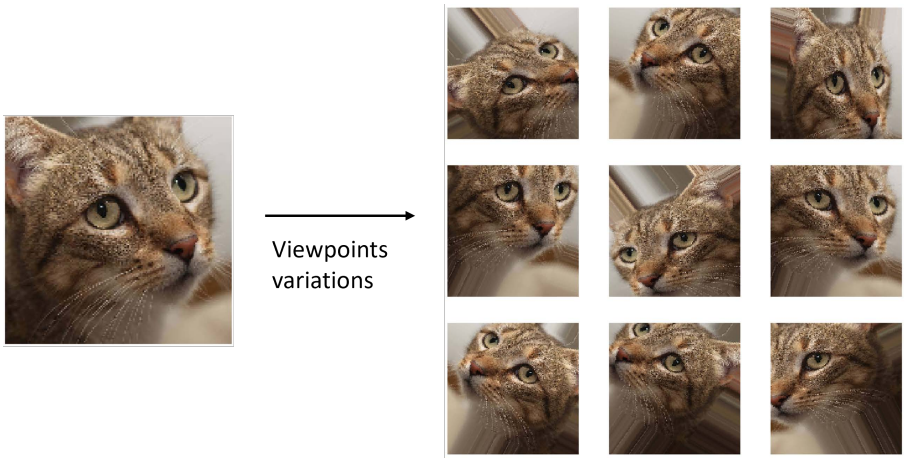


Figure 2.3. Viewpoints variations of an image adapted from [1].

Moreover, there are other issues to consider in the robustness of the algorithm in addition to the image point of view, as shown in Figures 2.4 and 2.5, such as illumination, deformations, occlusion, background clutter where the background of the image is similar to the class, or intraclass variation which defines the image variations that occur between different images of the same class, among others.

Due to all these situations, the design of a robust classification algorithm is complicated.

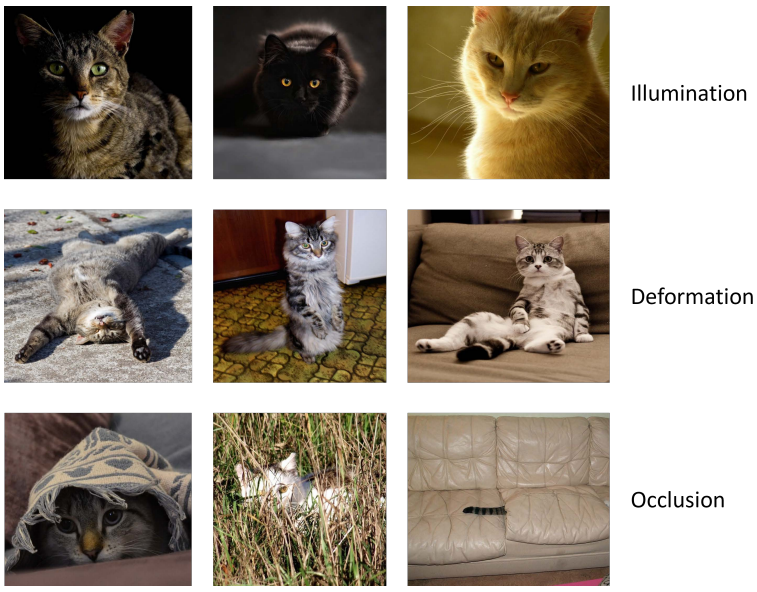


Figure 2.4. Images classification challenges: illumination, deformation, occlusion [1].

Figure 2.5. Background clutter and intraclass variation challenges [1].

Several attempts have been made to solve these challenges. At first, attempts were made to classify on the basis of human experiences, such as writing complex code rules to recognize different animals searching for specific features of an animal by finding edges and corners [44]. However, this type of approach is not generalizable to other objects with the same algorithm [1]. Therefore, a generalizable approach was needed, that is, one that can be trained with images of a single class and then use the same algorithm with minimal variations to train another class or classes.

A variety of models from the ML domain have been suggested to address the image classification task. Artificial Neural Networks, particularly Deep Learning, is the most advanced technique for this purpose [45, 11, 13]. Therefore, this thesis concentrates on the Deep Learning approach for image classification, as illustrated in Figure 1.3. Deep learning algorithms are able

to achieve excellent results due to the availability of a large amount of labeled data. Next, the importance of data is discussed in the following section.

2.3 Datasets

One of the key requirements for Machine Learning algorithms is to have enough data for training. In fact, one of the reasons why the use of Deep Learning models decreased at the end of the 1990s was largely due to the lack of enough data to train the models [45]. However, in recent years, technological advances (e.g., the Internet) have made it easier to produce a large number of datasets for different problems. In particular, datasets that involve images, from datasets with a few classes of objects to datasets with 22,000 classes (see Table 2.1).

Table 2.1. Some of the most common image datasets for computer vision tasks, such as detection (D), segmentation (S) and classification(C). Some datasets do not define a specific number for training and testing images, which is represented as Undefined (U).

Name	Creator	Year	Types of Images	Task	Categories	Training	Testing
MNIST [38]	New York University	1998	Digits	C	10	60 K	10 K
PASCAL VOC [46]	University of Oxford	2005	Miscellaneous	D	20	U	U
CIFAR-10 [47]	University of Toronto	2009	Miscellaneous	C	10	50 K	10 K
IMAGENET [12]	Stanford University	2010	Miscellaneous	D,S,C	1000	1.2 M	150 K
GTSDB [48]	Institut Fur Neuroinformatic	2010	Traffic signals	D	3	600	300
SUN [49]	National Science Foundation	2010	Scenarios	D	899	U	U
CARS [50]	Stanford University	2013	Cars	C	196	8144	8041
COCO [51]	Microsoft	2014	Miscellaneous	D,S	80	U	U
FASHION-MNIST [52]	Zalando Research	2017	Cloths	C	10	60 K	10 K
OBJECTNET [53]	MIT + IBM	2019	Miscellaneous	D	313	U	50 K

Another key requirement is *labeling*. For tasks such as object recognition, image classification, or scene classification, datasets need to be labeled. This means that the dataset contains the true values of what the data represent. The task of image labeling is indispensable, unfortunately, it is a tedious and time-consuming process that humans should perform to make sure the labels are correct (although there have been some attempts to automate this process [34]). An important thing to note is that some datasets are also useful for comparing the performance of

different approaches.

There are many datasets for image classification, as shown in Table 2.1. The dataset considered the state-of-the-art for classification tasks is ImageNet, which was constructed with public images from the Internet [45]. This dataset has more than 14 million high-resolution images with 22,000 categories and is continuously increasing. There is a compact version of this dataset that consists of 1,000 categories and approximately 1000 images for each category. Unlike other datasets, ImageNet provides three sets of images: approximately 1.2 million training images, 50,000 validation images, and 150,000 testing images. This compact version is one of the most popular datasets due to its use in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) since 2012 [12]. Furthermore, all of the most famous CNN architectures, with the exception of LeNet [54], were trained with the compact version of the ImageNet dataset. Recently, a new dataset, called ObjectNet, has been created with a new approach [53]. This dataset focuses on the object recognition task, which includes additional information about the objects, e.g. rotation, viewpoints, and backgrounds. It consists of 50,000 images and does not have a training-test division. All ObjectNet classes are objects that can be found in four different environments (kitchens, living rooms, bedrooms, and washrooms). ObjectNet has 313 classes, of which 113 overlap with ImageNet.

For other tasks such as detection or segmentation, more complex datasets are needed. For example, on a detection task, the target object must be localized and classified, whereas on the segmentation task, it is necessary to change the image into something more meaningful and easier to analyze [55]). Consequently, there are specific datasets such as: Cars [50], German Traffic Sign Detection Benchmark (GTSDB) [48], Scene UNunderstanding (SUN) [49], Common Objects in Context (COCO) [51] or ImageNet [45]. Because the scope of this thesis only reaches the image classification task, this research only uses two datasets: MNIST and COVIDx V7A , which are described below.

2.3.1 MNIST Dataset

The Modified National Institute of Standards and Technology (MNIST) dataset has been widely used in the image classification task and is considered a reference point. This dataset consists of 28×28 pixel grayscale images of handwritten digits with ten classes (numbers 0 to 9) as shown in Figure 2.6. This dataset has 60,000 training images and 10,000 testing images [38].

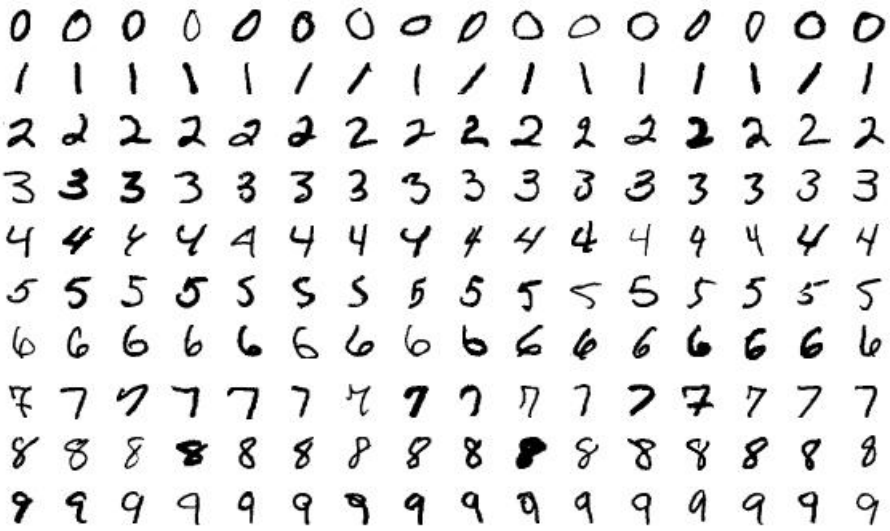


Figure 2.6. Examples of each digit in MNIST dataset.

2.3.2 COVIDx V7A Dataset

This thesis also used the open source COVIDx dataset [56], which is composed of different datasets such as: COVID-Chestxray dataset [24], COVID-19 Chest X-ray Dataset Initiative [57], COVID-19 Radiography Database [58], RSNA Pneumonia Detection Challenge [59], RSNA International COVID-19 Open Radiology Database (RICORD) [60], among others.

The version used for the experiments in this thesis was COVIDx V7A, which contains 16,690 images with three classes: pneumonia, healthy, and covid. Each image has a 1024×1024 pixel size in grayscale format. Figure 2.7 shows an example of each class in the COVIDx V7A dataset. In

summary, Table 2.2 shows the distribution of the dataset in terms of classes (training and testing). For training, we have 15,111 images spread over 5474 cases of pneumonia, 7966 cases of healthy images, and 1670 COVID images. The training dataset is divided into 80% training (12,089 images) and 20% for validation (3022 images). In the case of testing, we have 1579 images: 594 with pneumonia, 885 healthy, and 100 with COVID.

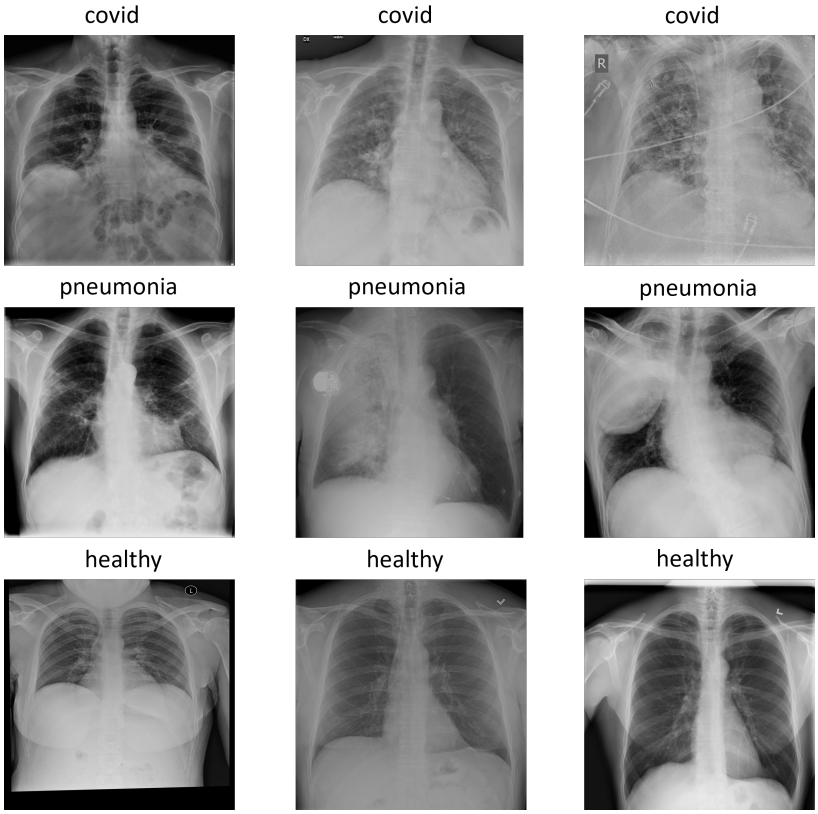


Figure 2.7. Examples of each class in COVIDx V7A dataset: covid, pneumonia and healthy.

Table 2.2. COVIDx V7A dataset classes.

Data	**Pneumonia**	**Healthy**	**COVID**	**Total**
train	5474	7966	1670	15,111
test	594	885	100	1579

2.4 Artificial Neural Networks

ANNs were first introduced back in 1943 by Warren McCulloch and Walter Pitts in the paper *A logical calculus of the ideas impermanent in nervous activity* [61]. They presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic [3].

Although scientists are still exploring the details of how the brain works, it is generally believed that the main computational element of the brain is the neuron. The neurons themselves are connected with a number of elements entering them called dendrites and an element leaving them called an axon. The neuron accepts the signals entering through the dendrites, performs a computation on those signals, and generates a signal on the axon. These input and output signals are termed activations. The axon of one neuron branches out and is connected to the dendrites of many other neurons. Connections between a branch of the axon and a dendrite are called synapses. So, Neural Networks take inspiration from the notion that a neuron's computation involves a weighted sum of the input values (see Figure 2.8).

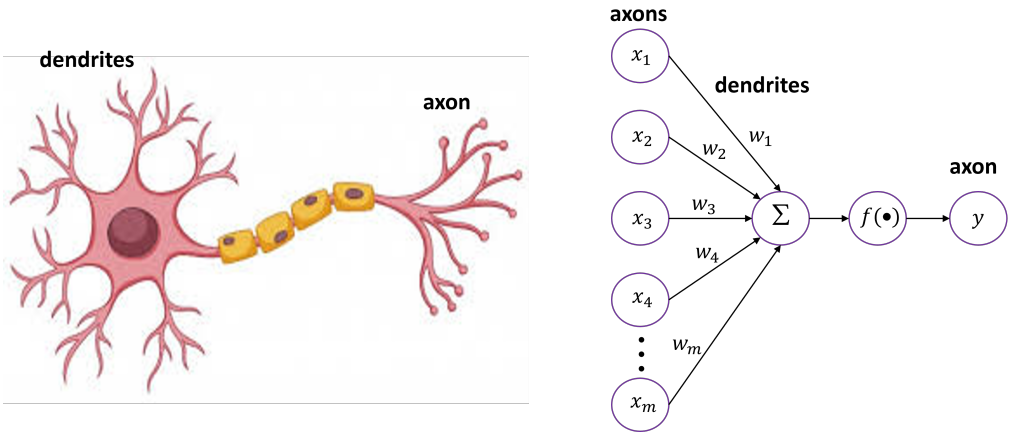


Figure 2.8. Comparision between a neuron (nerve cell) and an artificial neuron.

Therefore, the fundamental building block of every ANN is a single neuron also called a Perceptron. The perceptron takes an input or inputs x_i, then each input is multiplied by a corresponding weight w_i and adds the result of all these multiplications together. Then, the result of the sumatory is passed through a nonlinear function also known as an activation function, as

shown in Figure 2.9. The purpose of activation functions is to introduce non-linearities into the network.

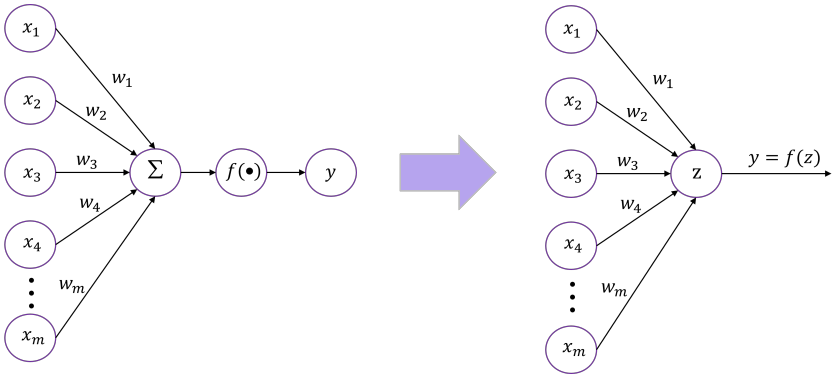


Figure 2.9. The Perceptron architecture adapted from [2].

So, the mathematical formulation is the single equation $y_j = f(\sum_i w_i x_i)$ where w_i, x_i and y_j are the weights, input activation and output activation, respectively, and $f(\bullet)$ is a non-linear function. The operation y_j defines how the perceptron propagates the information, and it repeats in each neuron. Also, the operation of the perceptron can be simplified as shown in Figure 2.9.

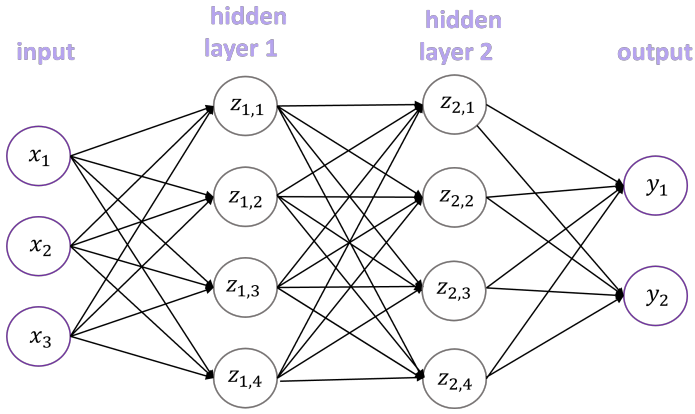


Figure 2.10. An example of ANN formed by an input layer, two hidden layers and one output layer, adapted from [2].

Now we can take a single neuron and start building it into something more complicated. If we want to build a multi-layered output Neural Network, we can stack layers as shown in Figure 2.10, where each connection represents a weight. The layers between the input and the output are also called hidden layers. Each neuron in each layer is connected to all the inputs of their previous

layer and the outputs, this type of connection is called Fully Connected (FC). Then, the output is going to be the next input of the next layer.

2.4.1 ANN Training for Image Classification

This subsection will explain briefly how ANNs are capable of classifying images. This classification ability is obtained through training, where a large number of input images are provided with their respective labels, also called true targets, to adjust many weights, which are responsible for extracting the descriptive characteristics of the objects. Therefore, the ANN model is a mathematical framework for learning representations from data.

In order to train an algorithm for a classification task, it is necessary to explain several concepts for the correct selection of the algorithm's parameters. Basically, the algorithm must have these four elements:

1. Input data: In the case of image classification, the input data are images.
2. Example of expected output (true target or ground true); in the image classification task, the expected outputs are labels.
3. A model to train means a network formed by many successive layers structured on top of each other where the information goes through, as shown Figure 2.11.
4. A metric or a way to measure whether the algorithm is doing a good job; this is necessary to determine the difference between the algorithm output and the expected output. The measure is used as a feedback signal to adjust the algorithm.

Each layer in Figure 2.11 is formed by a defined number of neurons and each layer is connected by weights. As mentioned above, the way the algorithm is capable of learning representation is due to its weights. Information about what a layer does to its input data is stored in the layer's weights. Initially, these weights have random values. But the goal is to find the right set of values for the weights of all layers in a network. Therefore, to know whether the weight values

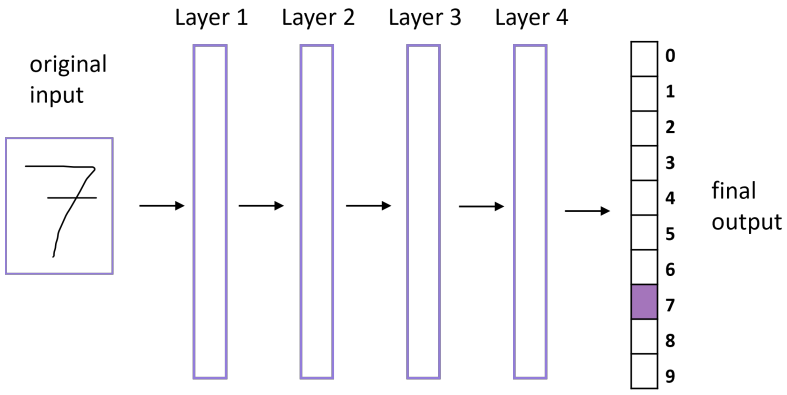


Figure 2.11. An example of Neural Network model for digit classification [3].

obtained are correct or not, the algorithm uses something called a loss function (J), also known as an objective function or a cost function. Therefore, the algorithm is going to try to minimize the loss function in order to find the right value for all weights.

So, the loss is a function of the network weights. and we can describe it as $J(\mathbf{W})$, where $\mathbf{W}$ is a set of all the weights in the network $\mathbf{W} = w_1, w_2, w_3, w_4, ...w_m$. To understand the function of the loss function, assume that the model has a loss function that depends only on two weights $J(w_0, w_1)$, and we plot all combinations of weights as shown in Figure 2.12. The algorithm then wants to find which set of weights gives it the smallest possible loss, that is, the lowest point in Figure 2.12.

So, the algorithm starts by randomly selecting an initial value w_0 and w_1 and computes the loss at that point. The algorithm then computes the gradient of the loss function defined by Equation 2.1 by backpropagation. The backpropagation algorithm determines for each single weight how much a small change in these weights affects the loss function, if it increases it or decreases it, and how we can use that to improve the loss.

$$gradient = \frac{\partial J\mathbf{W}}{\partial \mathbf{W}} \tag{2.1}$$

Therefore, the gradient is a function that evaluates the loss at all points. In addition, the gradient knows the direction of the highest point, that is, the values that increase the loss. Because

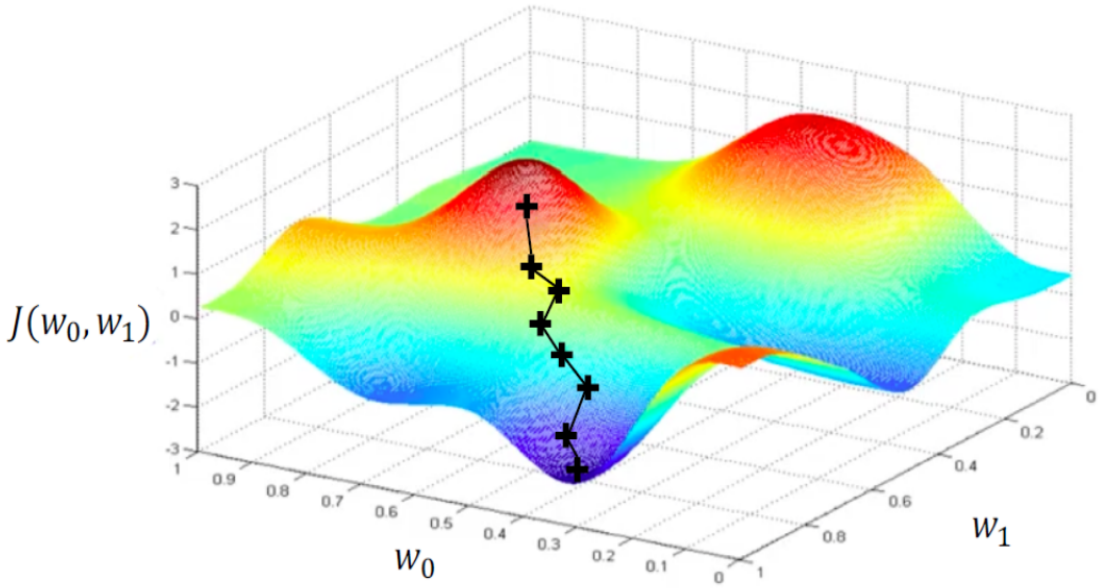


Figure 2.12. Example of the gradient descent algorithm, adapted from [2].

the objective is to decrease the loss, the algorithm negates the gradient value and updates the weights. Also, it is important to consider that computing the gradient can be very computationally expensive. One solution is that instead of computing the gradient over all the input data at once, the algorithm loads the input data in batches. Then, the algorithm moves the loss function values a small step, looking for the global minimum, as shown in Figure 2.12. This small step is called the learning rate (η). The formula that describes this operation is as follows.

$$\mathbf{W} = \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} \tag{2.2}$$

The algorithm then repeats this process over and over again, evaluates the model weights at the new location, and computes its gradient until it converges to the minimum value. Therefore, a challenge facing ANNs algorithms in training is the selection of η, because a small η converges slowly and could not converge to the global minimum in the function [3]. On the other hand, if the algorithm selects a large η, the function could start to diverge from the solution [3]. So, to select a good value η, we can try with different values η and see which offers the best result, or we can select adaptive learning rate algorithms, also known as optimizers, such as Stochastic

Gradient Descent, Adam, Adadelta, Adagrad, RMSProp among others [13, 11, 3]. Therefore, optimizing this algorithm is difficult because some networks have thousands of weights and found that determining the minimal value of the lost weight is a very complicated task.

In summary, the loss function's job is to take the network's predictions and the true target (the labeled data) and compute a loss score, calculating how well the network has done on a batch of input images. The algorithm then uses this score as a feedback signal to adjust the weight values slightly in a direction that will lower the loss score for the current example. This adjustment is the job of the optimizer, who implements what is called the backpropagation algorithm [62]. Figure 2.13 shows the training loop algorithm for an ANN model.

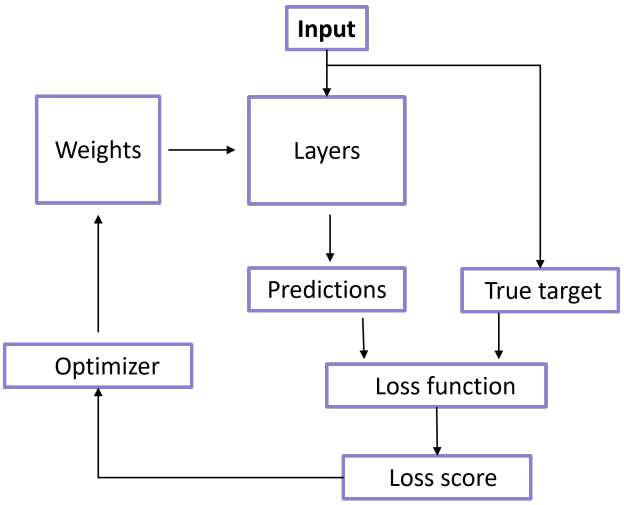


Figure 2.13. ANN training loop algorithm.

At first, the predicted output is far from the true target, and the loss score is high. But with every example processed by the network, the weights are adjusted slightly in the correct direction, and the loss score decreases. This training loop is repeated many times to minimize the loss function. Therefore, a trained network is a network that has minimal loss where the outputs are as close as they can to the true target.

It is important to mention the complexity of this training because a network can contain tens of millions of weights. At the same time, if we modify the value of one weight, this change will affect the behavior of all others, making it complicated for a person to predict the output of the algorithm.

2.4.2 ANNs Generalization

To develop an ANN model, it is necessary to understand two concepts: optimization and generalization. Optimization refers to the process of adjusting a model, finding the model parameters (weights), to get the best performance possible on the training data, whereas generalization refers to how well the trained model performs on data it has never seen before. So, in an ANN model, it is necessary to split the available data into three sets: training, validation, and testing. Then, the algorithm trains the model on the training data and evaluates the model on the validation data. At the beginning of training, optimization and generalization are correlated, as shown in Figure 2.14. The lower the loss in training data, the lower the loss in validation data. While this is happening, the model is said to be underfit: There is still progress to be made; the network has not yet modeled all relevant patterns in the training data. But after a certain number of iterations on the training data, generalization stops improving and then begins to degrade: the model is starting to overfit. That is, it is beginning to learn patterns that are specific to the training data, but that are misleading or irrelevant when it comes to new data.

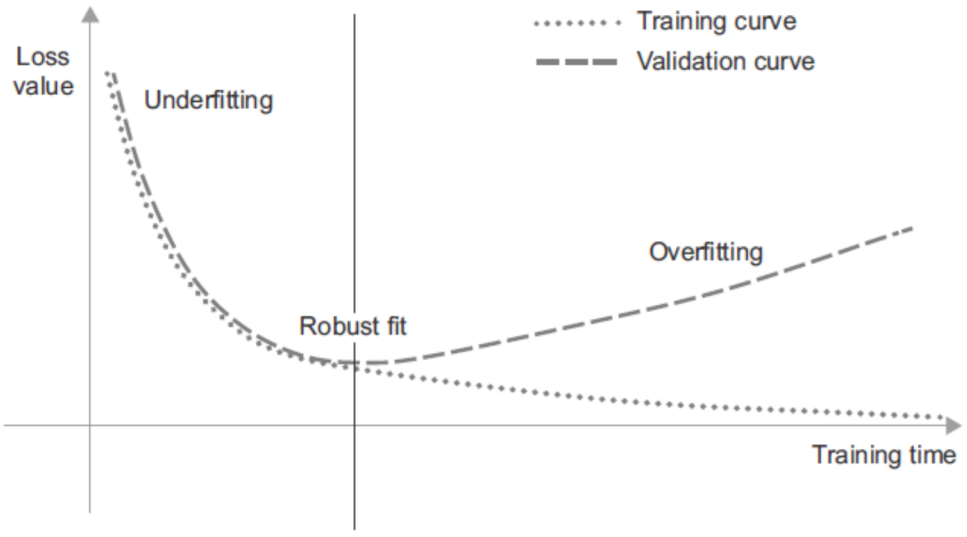


Figure 2.14. Example of the training behavior of a ANN algorithm adapted from [4].

The models tend to overfit in the training process, because the visual data is very complex, so it is necessary to design a complex model; this results in a high-dimensional model with a lot of

parameters to fit, and if we do not have enough training data, this overfitting is very common. To avoid the overfitting problem, there are regularization techniques. Regularization consists in constraining the model to simplify it and reduce the risk of overfitting. The most popular regularization techniques are:

- **Dropout**: this technique consists of randomly setting some neurons during training to 0, forcing the network not to rely on one path, changing weights and making different connections [63].
- **Early stopping:** this technique consist in detecting the point right after the validation loss starts to increase and the training loss still decreases, and using the weights at this point [64], as shown in Figure 2.14.

2.4.3 ANN Model Evaluation

Once the model is trained correctly, it is necessary to evaluate whether the model performs a good generalization on the new data. Therefore, here is where the algorithm uses a metric and the test data. A metric is a function that is used to judge the performance of the model. So, metric functions are similar to loss functions except that the results of evaluating a metric are not used when training the model. The metrics used in this thesis are defined below.

Confusion Matrix

The confusion matrix gives a comparison between the actual (test data) and the predicted values, and it is used to measure the performance of a classification model. It shows how many samples were correctly or incorrectly classified by the algorithm in each class as shown in Figure 2.3. The matrix has a size of N×N, where N is the number of classes or outputs. In Table 2.3 each row of the matrix represents the true class while each column represents the predicted class.

The correct classifications are on the diagonal of the matrix and the incorrect classifications are on the off diagonal of the matrix. Therefore, the total number of correct classifications is still

the sum of all elements on the diagonal and the total number of incorrect classifications is the sum of all of the off diagonal elements. For a binary classification problem the framework has two rows and two columns as shown Figure 2.3.

Table 2.3. Confusion matrix by a binary classification.

	Positive (Predicted)	**Negative (Predicted)**
Positive (Actual)	True Positive **(TP)**	False Negative **(FN)**
Negative (Actual)	False Positive **(FP)**	True Negative **(TN)**

In order to understand a confusion matrix for a binary classification, it is necessary to define four terms:

- True Positive **(TP)**: are the positive cases where the classifier correctly identified them.
- True Negative **(TN)**: are the negative cases where the classifier correctly identified them.
- False Negative **(FN)**: are the positive cases where the classifier incorrectly identified them as negative.
- False Positive **(FP)**: are the negative cases where the classifier incorrectly identified them as positive.

From these values that we obtain from the confusion matrix, we can obtain different metrics that will allow us to evaluate our model. One of the most used metrics are accuracy, precision, recall, F1-score which are explained below.

Accuracy: is the number of correct predictions divided by the total number of predictions, and it is defined by Equation 2.3.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

Precision: is the number of positive predictions divided by the total number of positive class values predicted in test data, and it is defined by Equation 2.4

$$Precision = \frac{TP}{TP + FP} \tag{2.4}$$

Recall: is the number of positive predictions divided by the number of positive class values in the test data, and it is defined by Equation 2.5. Also is called as Sensitivity.

$$Recall = \frac{TP}{TP + FN} \tag{2.5}$$

F1-score: The F1-score combines precision and recall using their harmonic mean, and it is defined by Equation 2.6.

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.6}$$

2.5 Chapter Summary

This chapter explains the challenges involved in an image classification task. It then gave a brief overview of some datasets used for the task, focusing on the two datasets used in this thesis. MNIST and COVID V7A. The last section explains the basics of Artificial Neural Networks and their application to the task of image classification, including the training, generalization, and evaluation stages of an ANN model. Finally, the metrics used in this thesis to measure the performance of our model were presented.

Chapter 3

Convolutional Neural Networks and Their Limitations

The preceding chapter discussed the image classification problem and the difficulties it presents. Additionally, it provided a review of the current solution approaches. As previously mentioned, the most effective way to address the image classification problem is through the use of Convolutional Neural Networks. This chapter will delve into the details of how this approach works and analyze its advantages and disadvantages.

3.1 CNN Architecture

Convolutional Neural Networks (CNNs) are based on the work of Hubel and Wiesel published in 1959 [37]. Their work describes the functioning of the cat's visual cortex and explains how animals observe things in a hierarchical way. At first, neurons perceive primitive features such as lines and edges, but as information progresses into the visual cortex, features become more complex, until specific shapes are formed (like a face or a car) [65]. Following this idea, CNNs are also composed of hierarchical layers. Each layer represents a series of operations that are applied to the input values. Then, the output of each layer becomes the new input value of the next layer.

As mentioned in Chapter 2, the first ANN architectures use only Fully Connected (FC) layers; this means that all neurons in each layer are connected to all neurons in the next layer, creating a bipartite graph [66]. A distinctive feature of a CNN is that each neuron in a layer is not connected to all neurons in the next layer, but uses partially connected layers and weight sharing, helping to decrease the number of parameters needed [54].

A CNN can be separated into two stages, the feature extraction stage and the classification stage, as shown in Fig. 3.1. In the feature extraction stage, the network employs convolutional layers, which use a mathematical operation called convolution. This operation is performed between the input value of the layer and different filters. It is important to note that the filter values are the network weights that are obtained automatically during the training phase. In the classification stage, the FC layer receives as an input a vector (flatten) coming from the feature stage, and the output are discrete values for the predicted classes, where the highest value is selected as the correct class. Unlike the feature extraction stage, the classification stage is not unique to CNNs because it is used in many ANNs architectures.

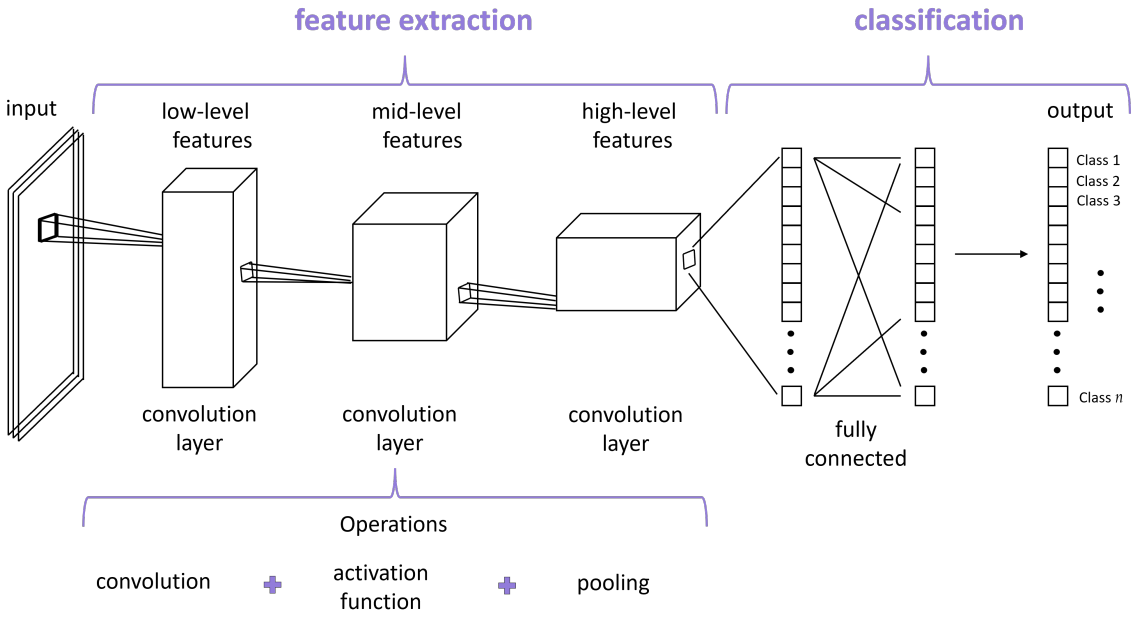


Figure 3.1. Basic CNN architecture for image classification.

3.1.1 Feature Extraction Stage

The feature extraction stage consists of several convolutional layers. As can be seen in Fig. 3.1, the input of the first convolutional layer is the original image, and its output becomes the input of the second convolutional layer and so on. Each convolutional layer uses three basic operations that are constantly repeated: convolution, activation function, and pooling. (see Fig. 3.1). For each operation, it is necessary to define the values of a set of hyperparameters such as: number of filters, filter size, stride value, pooling window size and padding value. A hyperparameter is a parameter of the algorithm; it must be set prior to training and remains constant during training. These concepts are explained below.

Convolution

In mathematics, *convolution* is an operation of two functions that produces a third one that expresses how the shape of one is modified by the other [4]. In order to understand the convolution operation, it is necessary to explain some definitions. The **receptive field** ($\mathbf{R}_{m,n}$) is a region of size $m \times n$, where each cell represents a pixel of the input images, as shown in Fig. 3.2. The **filter** or kernel ($\mathbf{W}_{a,b}$) is an array of numbers of size $a \times b$ (also known as weights or parameters) that are representations of features such as straight edges, simple colors, and curves. In Fig. 3.2, the filter is represented by the blue square which in this example covers an area of 3×3 pixels.

Then the convolution operation consists of multiplying each value of the **filter** with the values of the **receptive field** (the input image). Then, these values are summed to obtain one element of the output (known as **activation map** or **feature map**) ($\mathbf{O}_{c,d}$) as shown in Fig. 3.2. Each filter represents a different characteristic of the receptive field, so it is desirable to have enough filters to correctly describe the input image. For example, in Fig. 3.3 the input is a RGB image that employs two filters generating two feature maps; for example, one could represent horizontal lines and the other vertical lines. As we go through the network and through the convolutional layers, feature maps are obtained that represent more and more complex features.

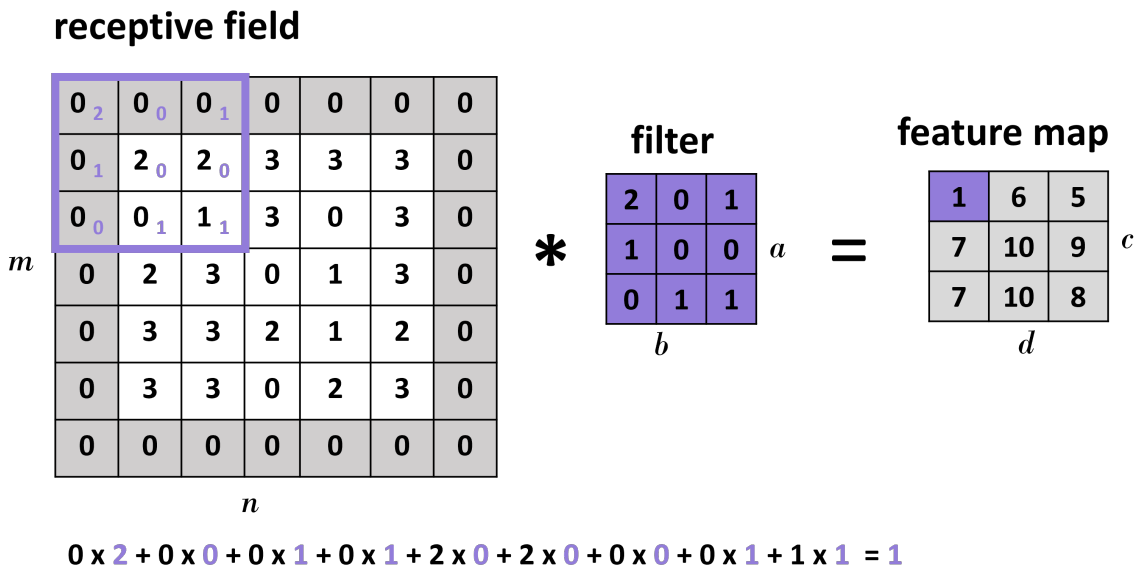


Figure 3.2. Example of the convolution operation. On the left hand side is the receptive field of size $m \times n$, the input. In the middle is the filter to be applied. On the right hand side is the feature map obtained after applying the filter to the input (receptive field). At the bottom is shown how an element of the feature map is calculated.

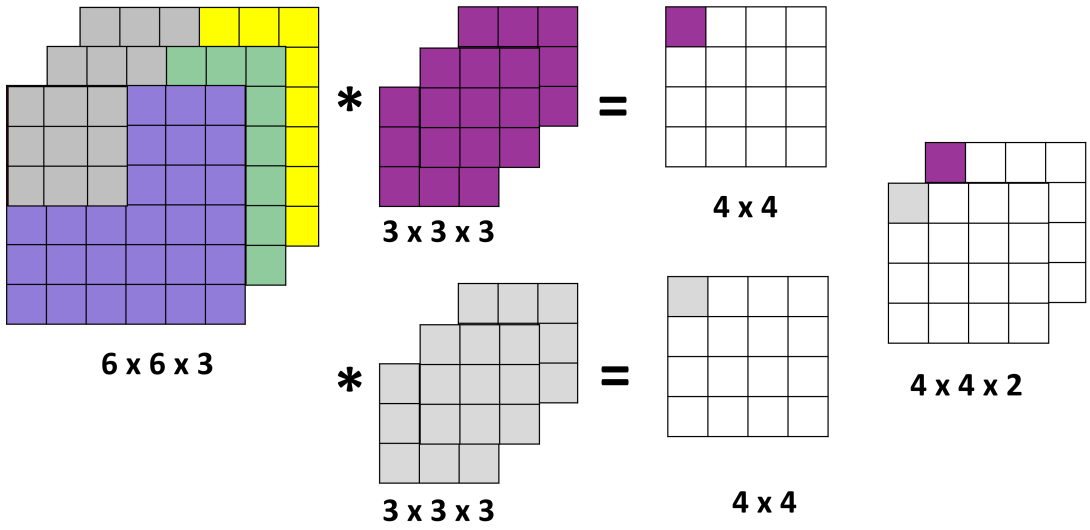


Figure 3.3. Example of a convolution operation on a RGB image with two filters generating two features maps of size $4 \times 4 \times 2$.

A simple convolution definition is as follows.

$$\mathbf{O}_{c,d} = \sum_{i=0}^{a-1} \sum_{j=0}^{b-1} \mathbf{W}_{i,j} \mathbf{R}_{i+sx,j+sy} \tag{3.1}$$

, where: $0 \leq x < c$, $0 \leq y < d$, $c = \frac{n-b+s}{s}$, $d = \frac{m-a+s}{s}$ and s is the stride value. Also, it is important to note that, for RGB images, the depth of the filter must be the same as the depth of the input, as shown in Fig. 3.3. There are two more hyperparameters that change the behavior of the network and the size of the output feature map. These hyperparemeters are stride and padding.

Stride The stride controls how the filter slides around the receptive field. In other words, the stride is the amount by which the filters shift. Fig. 3.4 shows an example with a receptive field of size 7×7 pixels and a 3×3 filter. After applying the first convolution operation and obtaining the first output pixel located in the upper left corner (pink pixel), Stride 1 allows the same filter to move one position to the right to apply the subsequent convolution and obtain the next output pixel (purple pixel). Consequently, the output dimension is reduced to a 5×5 feature map. In the second case, with a stride of two, the output dimension is further reduced to a 3×3 feature map. Also, it is important to note that the stride value is the same for the horizontal and vertical displacements of the filter. The stride value is selected so that the feature map is an integer value, not a real value.

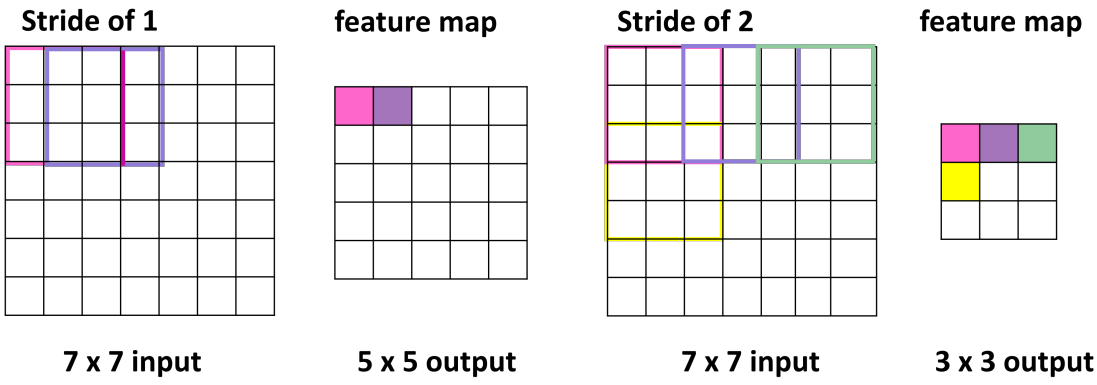


Figure 3.4. Example of how a different stride size can change the size of the feature map output.

Padding Sometimes it is desirable to preserve a specific feature map size. Thus, the padding hyperparameter refers to the number of pixels added to the receptive field to obtain a desired output size. Fig. 3.5 illustrates an example of padding. At the top, there is a 5×5 input with a 3×3 filter, and at the bottom the same input with the same filter size and zero padding. This yields a 7×7 input and a 3×3 output.

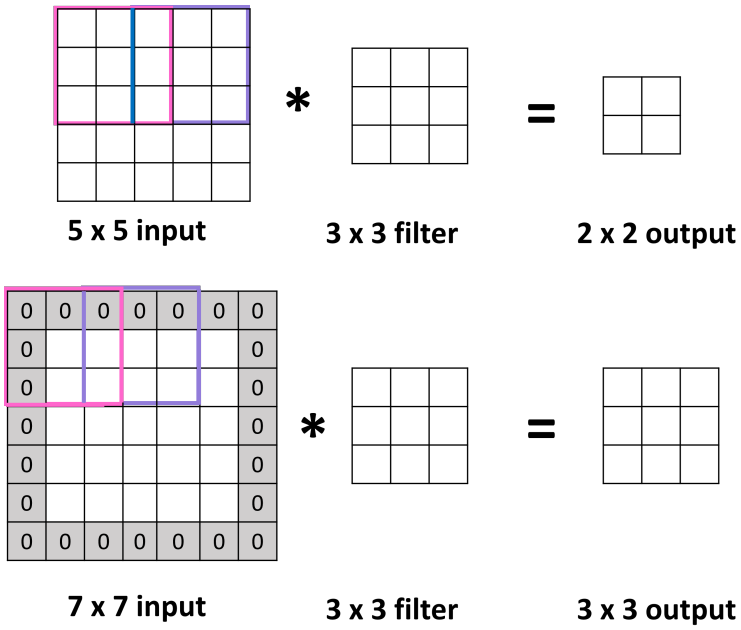


Figure 3.5. Example of zero padding hyperparameter, i.e. zeros around the border. Instead of zeros other values could be used, such as the average of the region.

Pooling

Usually after each convolution layer, the size of the input data is reduced, but the number of feature maps increases, as shown in Fig. 3.1. The number of feature maps is represented by the depth of each convolutional layer. This translates into an increase in the number of parameters and calculations that the network will have to handle. In order to process all this information, the pooling operation is used. This operation reduces the number of parameters or weights, preserving their important characteristics. In this operation, there are no estimations of weights. For this reason, some architectures do not consider pooling as a layer.

So, the pooling operation consists of choosing a region of the receptive field and downsampling that region by the selected method. The main methods are *max pooling* and *average pooling* as shown in Fig. 3.6. The output is a value that represents the most important part of the receptive field in the region. This is done in order to reduce the images to ease their processing without losing critical features to achieve good accuracy.

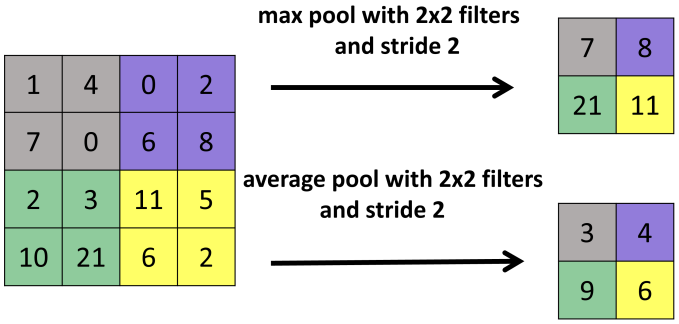


Figure 3.6. Pooling operation. Max pooling operation preserves the largest value inside of the selected size. Average pooling gets the average value inside the chosen size.

Activation Functions

Table 3.1. Examples of nonlinear Activation Functions

Name	**Function**
Sigmoid [67]	$y = \frac{1}{1+e^{-x}}$
Hyperbolic Tangent [68]	$y = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
ReLU [69]	$y = max(0, x)$
Leaky ReLU [70]	$y = max(\alpha x, x)$
Exponential LU [71]	$y = \{\frac{x, x \geq 0}{\alpha(e^{x}-1), x<0}$ where α is a small constant (e.g., 0.1)

After the convolutional operation, it is a convention to apply an activation function as shown in Fig. 3.1. This function, also called nonlinear function, is necessary for the model to be able to learn nonlinear functions. The first activation functions used were Hyperbolic Tangent and Sigmoid as shown Table 3.1. Currently the most used function is the Rectified Linear Unit (ReLU) due to the speed of its processing, which consists of changing all the negative values of the input

to zero and keeping the positive values. For improved accuracy, different variations of ReLU have also been proposed as shown Table 3.1.

3.1.2 Classification Stage

The classification stage is found in the last layer of all artificial neural networks. In this stage, the values of the last convolution layer are transformed into a single vector, then linear combinations and activation functions transform the input of this vector into another vector, and so on until another output vector is obtained (as shown in Fig. 3.1). The last vector of the network must have the same number of elements as the number of classes to be identified; each element represents the probability that the image belongs to that class. The probabilities are calculated by the last layer of the block using the softmax operation [4]. This operation ensures that the value of each element will be between 0 and 1, and the sum of all those elements is 1.

3.2 Main CNN Architectures

A CNN architecture can be defined as a sequence of convolutional layers with a different selection of hyperparameters in each layer, see Fig 3.1. One network is different from another simply by selecting a different configuration of one hyperparameter such as: the number of layers, number of neurons in each layer, number of filters and their size, stride value, padding value or the size of the pooling window [72]. It is important to emphasize that a slight change in the values completely changes the performance of the CNN. These networks handle millions of weights, thus it is not easy to predict the behavior of the network given a change. Also, each experiment demands a large computational time even with the use of Graphics Processing Units (GPUs). Therefore, the configuration of a CNN is still an active area of research because there is no formula that guarantees the correct selection of hyperparameters

Despite all these problems, several CNN architectures have been developed successfully, the most popular are the winners of the image classification task in the ImageNet Challenge (see

Table 3.2. Main CNNs architectures hyperparameters.

Hypeparameters	**LeNet [54]**	**AlexNet [14]**	**ZFNet [5]**	**VGG-16 [67]**	**GoogLeNet v1 [16]**	**ResNet 50 [17]**
Input size	28×28	227×227	224×224	224×224	224×224	224×224
# Conv layers	3	5	5	13	57	53
Filter sizes	5	3,5,11	3,5,7	3	1,3,5,7	1,3,7
# Filters	20, 50	96 - 384	96 - 384	64 - 512	16-384	64 - 2048
Stride	1	1,4	2	1	1,2	1,2
# FC layers	2	3	3	3	1	1
# Weights	60 k	61 M	62 M	138 M	7 M	25.5 M
ImageNet error	-	0.16	0.12	0.07	0.06	0.035

Section 2.3). A notable CNN architecture, named LeNet and created in 1989 by Yan LeCun [54], laid the foundations for the following networks and introduced new hyperparameters. This network was designed for the task of recognizing images on a grayscale of dimensions 28×28 pixels, and it was trained with the MNIST dataset [38]. Different configurations were designed, the most known version was LeNet-5. This configuration contains only three convolutional layers and two fully connected layers. In total, LeNet is made up of 60,000 weights, which makes it a not very complex network, compared to other architectures as shown on Table 3.2.

An example of the CNN complexity is the AlexNet architecture [14] compared to the ZFNet architecture [5], where just by changing the size of the convolution window filter in the input layer from 11×11 (as defined in AlexNet) to 7×7, the Top-5 error decreased from 16% (AlexNet) to 12% (ZFNet) as show in Table 3.2. Also, the Inception network by GoogLeNet was very innovative by proposing the option of not choosing just one filter size, but passing the image through different filter sizes. Then it combines all the output filter sizes in a single one, called DepthConcat [16]. At the same time, the VGG team designed a simple and elegant architecture [67], by using a filter of small size several times it intends to replace large filter sizes. For example, instead of a single 5×5 filter size it uses two filters of 3×3 size, obtaining satisfactory results, see Table 3.2. Due to its simplicity it is one of the most popular networks. Table 3.2 presents a summary of the most representative CNN architectures. The table shows the configuration of the hyperparameters used in each architecture. The last row of the table presents the approximate number of weights of each network, to show the complexity of these architectures [72].

Table 3.3. Most popular CNNs and their main contributions.

Network	Contribution
LeNet [54]	Designed for digit classification, average pooling operation, tanh as activation function.
AlexNet [14]	ReLU as activation function, max-pooling operation, parallel training, new regularization techniques: dropout and data augmentation.
ZFNet [5]	Changed the size of the filters in the convolutional input layer.
VGG Net [67]	Increased the depth of the network, use of smaller convolution filters.
GoogLeNet [16]	Use of inception modules, the network design occurs in parallel, nesting networks concept.
ResNet [17]	Residual blocks.

As can be seen in Table 3.2, the depth of the network and the number of layers is proportional to the classification error, where the more layers, the greater the accuracy. On the other hand, Table 3.3 summarizes the contributions of the most popular CNN architectures.

3.3 Literature Review on CNNs Limitations

Despite the application of CNNs in several computer vision tasks with outstanding results thanks to hardware technology such as GPUs, recent articles are beginning to report their limitations. When CNNs cannot achieve an acceptable classification result, we say that the CNNs have a limited performance. Several papers have identified some CNNs limitations, but most of them report only one type of limitation at most. One of the contribution of this thesis is to provide a description and analysis of all current CNNs limitations, and we propose to group them into the following categories:

- **Labeled data**: CNNs follow a supervised learning approach, which in turn requires a large amount of *labeled data*. As explained in Section 2.3, this is difficult to obtain in most real-world applications.

- **Translation invariance**: If the network was never trained with several rotations and translations of training images (i.e., to achieve *translation* and *rotation invariance*), then the network is likely to have a poor performance in the testing stage.

- **Adversarial examples**: Recent research works have shown that CNNs are vulnerable to misclassification errors if there is even a simple alteration in the input data. An altered image is known as *adversarial example*, and this alteration can be applied in different ways, from the perturbation (denoted as epsilon) of one pixel to the entire image.

- **Spatial relationship**: This limitation has to do with the way the network passes information from one layer to another. Since neurons do not consider the properties of features such as the spatial relationships, orientation or size, then CNNs do not respect the proportion of the objects found in the images.

Figure 3.7 illustrates the general idea about the categories of these limitations described above. In labeled data, we show the example of a retriever dog where it is necessary to label the elements of the image that we would like to classify. This task can only be performed by humans. In translation invariance, we show three images with the same dog, but these images have transformations to the original image, known by the CNNs, such as mirroring, rotation, cropping and translation. However, once the images are transformed then the CNNs are unable to classify them properly. In adversarial examples, a perturbation is added to the original image, then the perturbed image is not classified correctly. In spatial relationship, the elements of the figure are moved to different positions, for example the nose is at the left eye of the dog, and that eye was moved at the tongue position. These movements are unrealistic, but for the CNNs they are not. Therefore, the image will be classified correctly; Table 3.4 shows the results of the literature review, where each paper refers at least one limitation according to the four categories described above. This literature review focuses on papers published after the CNN's boom who started in 2012. Only old articles are included if they are referents in CNNs such as the first labeled datasets or essential architectures such as LeNet, created in the late 80s. The search engines used were Google Scholar, Springer, Scopus, ACM, and arXiv.

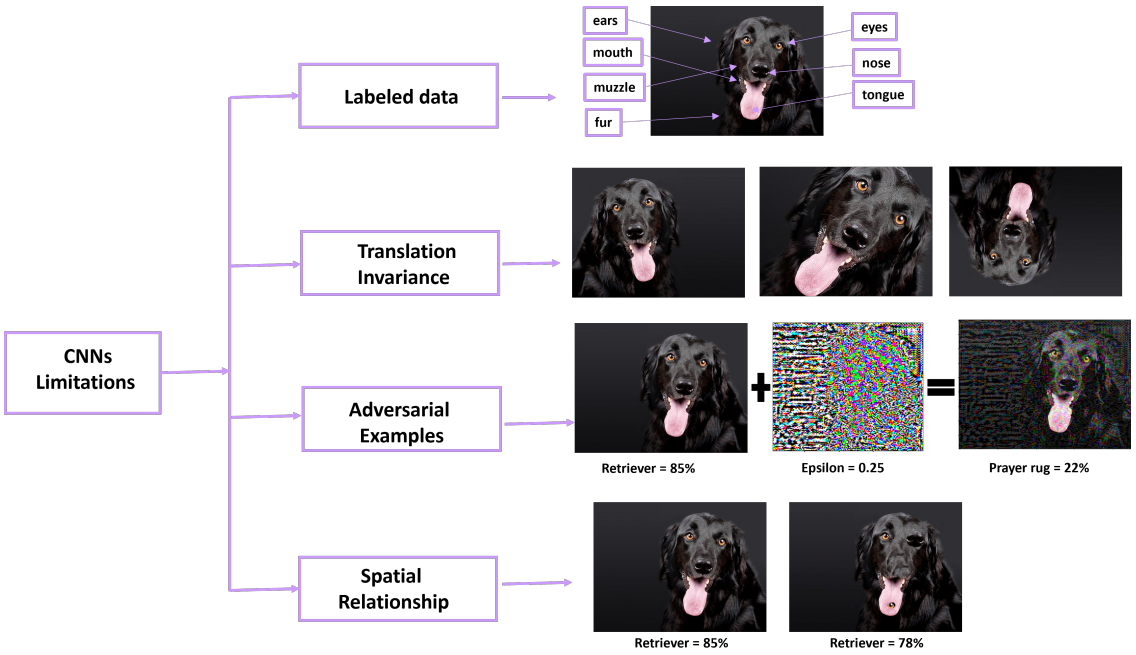


Figure 3.7. CNNs limitations with examples. It illustrates the four categories defined as CNNs limitations.

Table 3.4. Literature review on CNN limitations.

Paper	Labeled data	Translation invariance	Adversarial examples	Spatial relationship	Year
THIS THESIS	✓	✓	✓	✓	2023
Patrick, *et al.* [73]	✓			✓	2022
Alam , *et al.* [74]	✓				2022
Khodadadzadeh, *et al.* [75]	✓			✓	2021
Wang, *et al.* [76]				✓	2021
Steur, *et al.* [77]	✓				2021
Ma, *et al.* [78]				✓	2021
Luo, *et al.* [79]	✓	✓			2021
Fang , *et al.* [80]		✓		✓	2021
Hu, et al [81]	✓				2021
Ren, et al [82]			✓	✓	2020
Huang, et al [40]		✓		✓	2020
Sundaram, et al [41]	✓			✓	2020
Yang, et al [39]	✓			✓	2020

Continued on next page.

Paper	Labeled data	Translation invariance	Adversarial examples	Spatial relationship	Year
Brendel, et al [83]			✓		2020
Jalal, et al [84]	✓				2020
Jia, et al [42]				✓	2020
Wang, et al [43]				✓	2020
Patrick, et al [85]	✓	✓		✓	2019
Jayasundara, et al [86]	✓	✓			2019
Rosario, et al [87]	✓	✓			2019
Kruthika, et al [88]	✓			✓	2019
Rajasegar, et al [89]	✓	✓			2019
Hahn, et al [90]		✓		✓	2019
Su, et al [91]			✓	✓	2019
Zheng, et al [92]			✓		2019
Brown, et al [93]			✓		2019
Jayasundara, et al [86]	✓				2019
Kosiorek, et al [94]	✓				2019
Amer, et al [95]		✓			2019
Carlini, et al [96]			✓	✓	2018
Papernot, et al [97]		✓	✓		2018
Dong, et al [98]		✓	✓		2018
Xiao, et al [99]			✓		2018
Brown,et al [100]		✓	✓		2018
Hinton, et al [8]			✓	✓	2018
Frosst, et al [101]			✓	✓	2018
Afshar, et al [27]	✓	✓			2018
LaLonde, et al [102]	✓			✓	2018
Mukhometzianov, et al [103]			✓	✓	2018
Phaye, et al [104]		✓		✓	2018
O'Neill [105]		✓		✓	2018
Raghunathan, et al [106]			✓		2018
Wong, et al [107]			✓		2018
Guo, et al [108]			✓		2018
Wang, et al [109]			✓		2018
Liu, et al [110]			✓		2018

Continued on next page.

Paper	Labeled data	Translation invariance	Adversarial examples	Spatial relationship	Year
Lei, et al [111]			✓		2018
Eykholt,et al [112]			✓		2018
DARPA [34]	✓				2018
Siddiqui, et al [113]	✓				2018
Jaiswal, et al [114]				✓	2018
Xiang,et al [115]				✓	2018
Carlini, et al [116]		✓	✓		2017
Xi, et al [117]		✓		✓	2017
Papernot, et al [118]		✓	✓		2017
Sabour, et al [7]			✓	✓	2017
Kurakin,et al [36]			✓		2017
Sinha, et al [119]			✓		2017
Chen, et al [120]			✓		2017
Papernot, et al [121]		✓	✓		2016
Goodfellow, et al [122]				✓	2016
Papernot,et al [123]			✓		2016
Papernot,et al [124]			✓		2016
Papernot,et al [125]			✓		2016
Mohsen, et al [126]			✓		2016
Hinton, et al [127]		✓	✓	✓	2015
Simonyan, et al [67]	✓	✓			2015
Yosinski, et al [35]		✓			2015
Szegedy, et al [128]		✓	✓		2014
Zeiler, et al [5]	✓	✓			2014
Goodfellow, et al [129]	✓	✓			2014
Shorten, et al [130]		✓			2014
Goodfellow, et al [122]			✓		2014
Zeiler, et al [131]			✓		2013
Houben, et al [48]	✓				2013
Krause, et al [50]	✓				2013
Krizhevsky, et al [14]	✓	✓			2012
Geiger, et al [132]	✓				2012
Hinton, et al [133]			✓	✓	2011

Continued on next page.

Paper	Labeled data	Translation invariance	Adversarial examples	Spatial relationship	Year
Xiao, et al [49]	✓				2010
Everingham, et al [46]	✓				2010
Krizhevsky, et al [47]	✓				2009
LeCun, et al [134]	✓				1998
LeCun, et al [54]	✓	✓			1989

3.4 Possible Solutions to CNN Limitations

The previous section describes the four limitations that have been reported in different articles when using CNNs. These four limitations are: labeled data, translation invariance, adversarial example and spatial relationship. Next, these CNNs limitations are discussed along with possible solutions in order to overcome them.

3.4.1 Labeled Data

As explained in Section 2.3, one of the main problems to tackle is data availability. The more complex the computer vision task, the higher the quality and quantity of data needed for correct learning [12]. Intuitively, collecting a large number of images for training does not seem to be a problem because nowadays data is being generated everywhere all the time. Web platforms such as Youtube, Facebook, Google, Flickr, etc., can be used to retrieve data, images in particular. However, not only are these images required, but they must be *labeled*. The need for labeled data is because the CNN architectures make use of supervised learning, where the learning algorithms require to have data with the correct labels for the training stage [11].

For image classification, labeling consists in determining the class that belongs to the object within that image. Sometimes an image may contain several objects that belong to several classes [12, 135]. It is important to emphasize that the labeling task is normally done by humans to guarantee a correct labeling, and even then there could be some errors. This makes the labeling

task time-consuming requiring a large number of people focused on this task. For example, the ImageNet dataset required 49,000 workers from 167 countries and it took three years for labeling one billion images [12].

So, the first CNN limitation is about the large number of images needed to train these networks. The main CNN architectures coincide that the larger the number of images used during training, the better the resulting accuracy. These models require around 10^9 or 10^{10} labeled data to achieve a good performance [34]. Therefore, training a CNN from scratch is a complicated, expensive and time-consuming job.

As a result of this limitation, there are several free labeled datasets for training, such as ImageNet, MNIST, Pascal, COCO, among others, as mentioned in Section 2.3. However, when the CNN application is specialized, customized, or new, the existing datasets could be useless. So, new strategies are required in order to train CNNs with a limited number of labeled images. This limitation is reported in several research papers as shown in Table 3.4

One of the most used strategies to overcome this limitation is a process called **transfer learning** [13]. The main idea is to use the optimized weights of an existing CNN model (e.g., AlexNet [14], GoogLeNet [16], ResNet [17], etc.), to perform classification with different classes using a smaller number of input images. The changes for implementing this strategy are made in the classification stage on the last fully connected layer, as shown in Fig. 3.1. The training only changes the values of the weights on this stage. The number of neurons in the last layer has to be equal to that of the classes for the new task, this process is called *fine tuning*. This makes the training much faster, requires fewer input data and less computational effort. Also, due to its successful results, it is the most widely used technique in real-world applications. For example, for classifying fish and other marine species in underwater videos [84, 113], or for the diagnosis of plant diseases to save crops [19].

On the other hand, there are attempts to do the labelling task automatically. For example, the Defense Advanced Research Projects Agency (DARPA) aware of this limitation launched the *Learning with Less Labels* (LwLL) program in August 2018 [34], and still under development. The first part of this program was about the developing of algorithms with the goal of reducing

the amount of labeled data needed to train a model from scratch by at least a factor of 10^6 and adapt it to new environments with only hundreds of labeled data. Another LwLL goal is that the algorithms can use all the amount of unlabelled data available and the algorithm autonomously selects specific examples for labeling, i.e., clustering. Algorithms can create data, but cannot create labels. To achieve these ambitious goals, advanced methods are expected in meta-learning, automated transfer learning, reinforcement learning, active learning, unsupervised or semi-supervised learning, and k-shot learning [11]. The second part of this proposal was to test the limits of the amount of labeled data necessary to solve different Machine Learning problems. The behavior of the models will be analyzed before each problem, because DARPA is looking for mathematical theorems that could explain the relationship between the number of input data and the correct learning of the model [34].

3.4.2 Translation Invariance

The second CNN limitation is the translation invariance, this refers to the problem where an object with a slight change of position or orientation is not correctly classified by the network. Therefore, it requires different points of views of each labeled object.

Zeiler et. al. performed an experiment with the ZFNet, see Fig. 3.8 [5], where five randomly images from ImageNet were used: Lawn Mower, Shin-Tzu, African Crocodile, African Grey Parrot, and Entertainment Center. They showed that the network was trained with the original image, and then just by doing a small image transformations (translation, scaling and rotation) on the test stage, the accuracy (P true class) of the network can be dramatically affected.

To overcome this limitation, researchers use **data augmentation** based on image manipulations, that consists of artificially increasing the number of images in the dataset by using different augmentation techniques, such as geometric transformations (e.g., flipping, rotation, scaling, cropping, translation, Gaussian noise addition) or advanced augmentation techniques based on Deep Learning (e.g., conditional generative adversarial networks (GAN), neural style transfer, adversarial training) [130]. Data augmentation has been used in the training of the most popular CNNs architectures (see Section 3.2), since it was successfully introduced in AlexNet [14]. Table 3.5

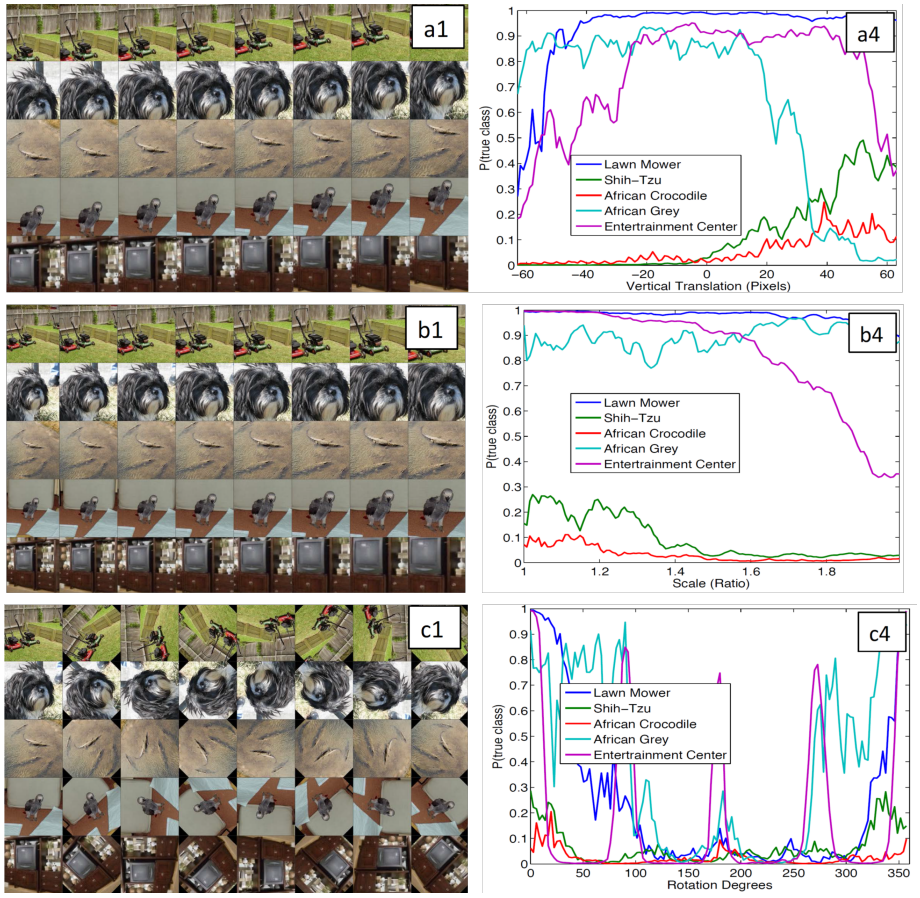


Figure 3.8. Example of the translation invariance limitation (adapted from [5]). In the first column three transformation techniques were applied to five sample images: (a1) vertical translation, (b1) scaling, and (c1) rotation. In the second column, the probability of the true label for each image, as the image is transformed, is shown. The more the transformation the lower the probability of the true class.

mentions the data augmentation techniques used by popular CNNs architectures. Also, many researchers report good results augmenting data for testing images as well [130].

Data augmentation can also help on some problems related to datasets. For example, alleviating class imbalance, which is a problem where a dataset is primarily composed of examples from one class. In addition, data augmentation helps to downsample images with high resolution, such as HD or 4K, making them computationally easier to process [130]. At the same time, data augmentation generates more labeled data increasing the size of the dataset, which in turn helps to prevent overfitting because we obtain more training and testing data to achieve

Table 3.5. CNN data augmentation techniques.

Architecture	Data augmentation technique
AlexNet [14]	Images generated by translations, horizontal reflections and altering the intensities of the RGB channels.
ZFNet [5]	Images generated by producing multiple different image crops and flips of each training example to boost the training set size.
VGG [67]	Images generated by cropping with random horizontal flipping and random RGB color shifting, each training image is individually re-scaled using random sampling.
GoogLeNet [16]	Images generated by resizing the original image to 4 scales (256, 288, 320 and 352), and also by generating their mirrored versions.
ResNet [17]	The original image is first resized, then it is cropped and flipped.

generalization [130].

Nevertheless, the data augmentation process requires great computational effort given the amount of data generated. This computational effort translates into additional memory and computational constraints. There are two options for data augmentation: offline or online (i.e., before or during training). Offline augmentation transforms data beforehand and stores it in computer memory, which can be problematic for large datasets. On the contrary, online augmentation transforms data on the fly during training. This option can save memory, but will result in slower training. Also, it is important to note that for some problems data augmentation should not be performed because it may significantly change the content of the information (e.g., voice data, disease images, among others).

3.4.3 Adversarial Examples

Researchers have discovered an intriguing phenomenon called **adversarial example** [128]. This phenomenon consists of easily cheating CNNs with a test image slightly modified (known as adversarial example), that forces the architecture to misclassify with high accuracy. A simple example was shown in Fig. 3.7, where the attacker adds a small perturbation (noise ϵ) that has

been designed to make the image to be recognized as a rug instead of a dog. When an adversarial example is applied on a Machine Learning model, it is called an adversarial attack [82].

There are different attack scenarios, and we classify them into four categories, see Table 3.6. The category p-norm refers to the distance function, known as p-norm. This norm measures the distance between the original image and the perturbed image with a loss function L_p, where $p = 0, 1, 2, \infty$ [18, 136, 137, 82]. The images are perturbed by adding noise through the parameter ϵ. The objective is to maximize the classification error in order to make the CNNs fail by minimizing the difference between the original image and the perturbed image; being imperceptible to the human eye [137].

Table 3.6. Classification of adversarial attacks scenarios by category and name.

Category	**Type**	**Description**
p-norm	L_0, L_1, L_2, L_∞	This norm is used to evaluate the difference between a perturbed input and the original input.
Data feed	Digital	The adversary has direct access to the actual data fed into the model.
	Physical	The adversary does not have direct access to the digital representation of the provided model.
Goal	Untargeted	The goal is to cause the classifier to predict any incorrect label.
	Target	The goal is to cause the classifier to predict some specific label.
Knowledge	White box	The adversary has full knowledge of the model.
	Gray box	The adversary does not know very much about the model but can probe the model.
	Black box	The adversary has limited or no knowledge about the model under attack and is not allowed to probe or query the model.

Concerning the Data Feed category, it is common to assume that attacks are delivered to the

network digitally. However, there are adversarial attacks that can be brought into the real world. For example, Kurakin et al. performed an exciting experiment where they took some clean images from the ImageNet dataset, and they used it to generate adversarial examples [36]. Those images were printed out, and then they used the TensorFlow Camera Demo app on a standard cellphone, which is based on the GoogLeNet architecture [16], to classify the printed images. This resulted in a large number of adversarial examples classified incorrectly even when they are perceived by the camera. This shows that these attacks are robust because they are capable to fool the system, despite being printed.

The Goal category contains two types: untargeted and target; where the first one the goal is that the classifier predict incorrect labels and for the latter the goal is to predict some specific label. For example, a successfully adversarial attack is the adversarial patch based on the Expectation Over Transformation (EOT) algorithm [100, 138], which can be used to attack any scene, without the need of knowing the other items in the scene and causes the classifier to output a targeted class. They simply place the patch physically in the scene to be attacked. They used the VGG network, first they select an image with a banana and get a 97% accuracy, then they include the patch designed to classify as a toaster in the scene and get a 99% accuracy in the toaster class [100]. Another example of target attack and physical data feed is the one that mimics graffiti stains on traffic signs. The goal is to misclassify a *stop sign* into a *speed limit 45 sign* [112]. They managed to get a classification system to fail against a proven real-time architecture, namely You Only Look Once (YOLO) [139]. The disturbance was included in the form of a black and white sticker, the attack was made to a real stop sign causing a failure in the classification of 100% in a controlled environment. The same experiment with a stop signal was performed, but now the images were captured on video through a moving vehicle, where they obtained a 84.8% classification error [112].

The Knowledge category, that refers to the model, has three types: white box, gray box and black box; where the white box has full knowledge about the model and the black box has limited or no knowledge about the model, and the gray box is something in between. In addition, this category presents an interesting phenomenon called transferability [124], this means that a designed adversarial attack for a specific model can attack another architecture without knowing

the model. That means a white box attack could work as black box attack because not only affects a CNNs models but also affects different Machine Learning techniques, such as SVMs or decision trees [136, 82, 98, 137].

Creating adversarial attacks in Machine Learning is a trial-and-error process. Table 3.7 presents a comparison of different adversarial attacks algorithms. The first column contains the name of the algorithm, the second column is the Goal category described in Table 3.6. The third column refers to the Knowledge category also described in Table 3.6. The columns Noise and Geometric correspond to algorithms that use different strategies to create an adversarial example. Some algorithms change the elements of the image, where it could be adding noise to specific areas of an image. Others use geometric transformations, e.g., rotations or translations, to induce misclassifications. Now, the columns p-norm, Image and Universal refer to algorithms designed to fool the networks by perturbing an image. The perturbation p-norm adds noise and measures the difference between the perturbed image and the original one. The perturbation by Image means perturbing only one specific image. In contrast, Universal perturbation means adding some noise to any image instead on focusing on a single one [140]. Finally, the last column indicates the methods used to generate the adversarial attacks, i.e., one-step gradient, iterative, and optimization.

Table 3.7. Comparison among different adversarial attacks algorithms.

Algorithm	**Goal**	**Knowledge**	**Noise**	**Geometric**	**p-norm**	**Image**	**Universal**	**Method**
FGSM [129]	Both	White box	✓		L_∞	✓		one-step
L-BFGS [128]	Both	White box	✓		L_2,L_∞	✓		one-step
BIM [36]	Untarget	White box	✓		L_∞	✓		iterative
Momentum [98]	Untarget	White box	✓		L_∞	✓		iterative
C&W[116]	Both	White box	✓		L_0,L_2,L_∞	✓		iterative
JSMA [123]	Both	White box	✓		L_0	✓		iterative
DeepFool [126]	Both	White box	✓		L_0,L_1,L_∞		✓	iterative
Universal [140]	Untarget	White box	✓		L_2,L_∞	✓		iterative
One-pixel [91]	Untarget	Black box	✓		L_0	✓		iterative
UPSET [141]	Target	Black box	✓		L_∞		✓	iterative
ManiFool[142]	Both	White box		✓	own	✓		optimization
EOT [138]	Target	White box		✓	own	✓		optimization

So far, it does not exist a reliable technique that guarantees a complete defense against an

adversarial attack. There are several novel adversarial defensive techniques, but these techniques only work for specific attacks [131, 121, 97, 118]. The defense techniques can be classified into three approaches (see Table 3.8): Guardians, Design and External. The first refers to techniques that do not interact with the model, but the defense is performed by modifying the training or the input images in the testing [143, 144, 145, 146, 147, 148, 144, 149]. The second, and most popular technique, consists of changing the network architecture and the training data or loss function [150, 129, 151, 152, 153, 154, 127, 125, 155, 156]. The third approach uses external models to its architecture when classifying unseen examples [157, 158, 159, 160].

Adversarial training is one of the algorithms that offer the best results of all defense techniques [129]. The basic idea is to generate a lot of adversarial examples and training the model so that it does not fail if it is attacked. However, adversarial training brings a high computational cost. Another algorithm with good results is defensive distillation, which proposes a technique that can use any artificial neural network and increases its robustness by reducing the rate of attack success [125, 127]. The idea is to use multiple classifiers instead of just one and have them vote the prediction (ensemble), but it requires more computational resources in order to obtain a proper defense against attackers.

Table 3.8. Examples of adversarial defensive algorithms.

Approach	**Algorithms**
Guardians	Data compression [143, 144], Foveation [145], Data Randomization. [146], PixelDefend [147], SafetyNet [148], Bit-Depth [144], Adaptive Noise [149]
Design	Deep Contractive Networks [150], Adversarial Training [129], Gradient Training [151], Gradient Regularization [152], Robust Training [153] , Stochastic Pruning [154], Defensive Distillation [127, 125] , Parseval Networks [155], DeepCloak[156]
External	Defense Against Universal Perturbation [157], GAN-Based [158], Feature Squeezing [159], MagNet [160]

Each strategy tested so far is defeated because networks are not adaptive, that is, they can block only one type of attack, but if the attacker knows the defense technique, it is extremely easy to

improve the attack. For these reasons, designing an adaptive defense technique is an important area of current research. For now, the adversarial training technique is the most successful. However, it does not work in all attack scenarios and requires a high computational cost to design it. For these reasons, the adversarial example is another research area for the scientific community, where some competitions focus on accelerating investigations and measuring the robustness of current defense algorithms. Examples of these competitions are: the Adversarial Vision Challenge [83] and the Adversarial Attacks and Defenses Competition[161], among others.

As mentioned above, CNNs have become popular because of their high accuracy, so these networks are being incorporated into real-world systems. For instance, in autonomous driving vision systems where it is important to recognize pedestrians, traffic signals, and vehicles. Thus, it is expected that these networks will be robust to small input perturbations. However, it appears that CNNs may fail in some situations. At the moment, the cause of misclassifying these types of examples is still unknown. Some researchers think that it could be due to the non-linearity of the models and the overfitting [129, 136]. Other researchers have pointed out that it could be a consequence of the high geometry of the data and the lack of training data [82].

3.4.4 Spatial Relationship

The fourth limitation is related to the spatial relationship, as shown in Fig. 3.7, where CNN classifies the two images as a golden retriever. Despite having all the elements of a dog (i.e. two eyes, two ears, one nose, a tongue), for a human the second image should not make sense. This is due to common sense that tells us that the nose is below the eyes and that the eyes must be side by side and not on the tongue. Therefore, the networks must not only detect and classify objects within an image, they must also learn the spatial relationships among features [40]. Due to the results presented by CNNs, it can be said that they tend to memorize the data rather than understand it [39].

Researchers explain that one reason why the image is wrongly classified is due to the routing [7, 8], that refers to how the network passes information from one layer to another. Because neurons are activated based on the opportunity to detect specific features, neurons do not consider

feature properties such as the spatial relationship, orientation, or size of objects. As explained in Section 3.1, after convolution in most CNN architectures, the size of the input data is reduced, but the number of features increases; see Fig. 3.1. To process all these features, the pooling operation is used for routing. Therefore, the pooling operation contributes to the limitation of spatial relationships. In Section 3.1, it was explained how the pooling layers are successfully applied within the architecture. However, several researchers argue that the use of this operation presents some disadvantages such as the following.

- If the object to be detected is very small, after the max pooling operation the size of the pixel will be further reduced, making it more difficult to detect [41].
- By reducing the number of parameters from one layer to another, important information about the spatial relationship of the components is lost and the focus will only be on the presence or absence of features [40].
- Losing the spatial relationship of objects requires more training images and will force the network to use tools such as data augmentation to improve its performance [104].
- The pooling operation can provide a little translation invariance, but will lose the precise location information of features [39].

To overcome this limitation, new approaches are required. Hinton proposes the concept of equivariance [7], where if the input image is rotated, the neural network changes and adapts all the features of the spatial relationship to the movement of the input image. Hinton argues that the brain does the opposite to computer rendering programs [133, 7, 8]. He calls it inverse graphics, where the visual information received by the eyes is deconstructed like a hierarchical representation of the world around us by the brain, and will try to match it with already learned patterns and relationships stored in the brain. This is how recognition happens, and the key idea is that the representation of objects in the brain does not depend on the view angle. Hinton's idea is based on the hypothesis that a small child does not need to see thousands of images of an object to recognize it when it is seen from the back or rotated because the brain internally performs all these operations [162]. The following chapter describes a new approach capable of addressing this shortcoming.

3.5 Chapter Summary

As the objective of this thesis is to create a computational model combining two architectures: CNN and CapsNet. This chapter focuses on CNNs, which are highlighted by detecting features. First, it explains the internal architecture of these networks and the hyperparameters needed to configure them. It then explains the main convolutional networks that exist and their advantages. One of the contributions of this thesis is the identification and grouping of these limitations into four groups: labeled data, translation invariance, adversarial attacks, and spatial relationship. For these reasons, the third section presents a deep analysis of their limitations and explains that finding a solution to these limitations is not easy, because designing a CNN is complicated. It is still a handmade process, because no formula ensures the correct functioning of one architecture. CNNs are formed by different hyperparameters (i.e., filter size, nonlinear function, size, and number of layers) which modify the behavior of these networks. Due to all these limitations, new solutions are needed. In the next chapter, we will present a new architecture capable of addressing all these limitations.

Chapter 4

Capsule Networks as an Alternative Solution

Due to the limitations of CNNs described in the previous chapter, new approaches have been proposed. This chapter covers an approach, still under research, as an alternative to overcome such limitations. Capsule networks (CapsNets) were introduced by Geoffrey Hinton and his students Sara Sabour and Nicholas Frost in 2017 [7]. They explain that the brain is organized into modules that can be thought of as capsules. These capsules are adept at handling different types of visual stimuli such as pose (position, size, orientation), deformation, speed, pitch, texture, etc. They also mention that the brain must have a mechanism to "route" low-level visual information to the capsule it deems best suited to handle it. A CapsNet is organized in several layers, much like a normal neural network. The capsules in the lowest layer are called primary capsules. Each of them receives as input a small region of the image (receptive field). It tries to detect the presence and position of a particular pattern, e.g. a rectangle. Then, the capsules located in higher layers, called routing capsules, detect larger and more complex objects, such as ships or a face. CapsNets are thus a type of ANN, and their goal is to improve the way that ANNs pass information through their layers. In addition to that, Hinton's team also published an algorithm, called dynamic routing, that allows one to train this new network because now the information is in vector form, instead of having scalar values.

A capsule is a group of neurons that code the probabilities of feature detection, and they

output a small vector of highly informative outputs. Each capsule has an array of 8 values, which is also called a vector. Now these vectors or capsules are the new pixels. Previously, with a normal pixel, we only have a scalar number (0-255), whereas a capsule can store 8 values per pixel, which means that capsule can store more information. The stored information is all the data necessary to describe an image, such as shape, position, rotation, color, or size. In Sabour's paper these descriptors are called instantiation parameters [7]. With more complex images the instantiation parameters can include pose (position, size, orientation), deformation, velocity, albedo, hue, texture, etc. In addition, each capsule has two components: magnitude and orientation. The magnitude represents the probability that the entity exists, while the orientation represents the instantiating parameters or properties of the entity.

As mentioned in the previous section, CNNs are the state of the art in image classification; however, they also have several limitations. One of the most important limitations is related to the spatial relationship between the characteristics of an image. The max-pooling operation in CNNs generates the problem of data invariance. This refers to the fact that small changes in orientation or position in the input features will not produce a change in the output because CNNs focus only on the absence or presence of features. Whereas CapsNets look for equivariance where changes in the input image generate changes in the model output. CapsNets encode the detection probability of a feature as the magnitude of its output vector. The state of the detected feature is encoded as the direction to which that vector points ("instantiation parameters"). Therefore, when the detected feature moves across the image or its state changes in some way, the probability remains the same (the length of the vector does not change), but its orientation changes.

For example, Figure 3.7, where CNN classifies the two images as a golden retriever because they detected its two eyes, two ears, one nose and a tongue; but the spatial distribution of these elements and their relationship between them are not really considered by CNN; because the network extracts certain features of the image such as eyes, ears, nose, etc. The higher-level layers will then combine those features and check if all of those features were found in the picture, regardless of order. On the other hand, CapsNets will detect if the image elements are slightly modified (for example, translated, rotated, or resized), because the capsules will produce a vector of similar length, but with different orientation. This order is determined during training

(dynamic routing), when the network learns not only what features to look for, but also what their relationships to one another should be. For instance, it might be learned that the nose should be between the two eyes and the mouth should be below that. Images with these features in specific order will then be classified as a dog and everything else will be rejected. This means that the system will only detect a face if the features detected by the capsules are present in the correct order.

4.1 Computing a Capsule and a Neuron

In this section, the principal differences between computing a neuron and a capsule are presented. Figure 4.1 shows the connection to compute a neuron.

So, the steps to follow for compute a neuron are:

1. Multiply the input scalars $(x_1, x_2, \dots x_n)$ with the weighted connections $(w_1, w_2, \dots w_n)$ between the neurons.
2. Compute the weighted sum of the input scalars, i.e. $\sum_i^n x_i w_i$.
3. Apply an activation function (f) to the scalar values to get the output, $y_j = f(\sum_i^n x_i w_i)$.

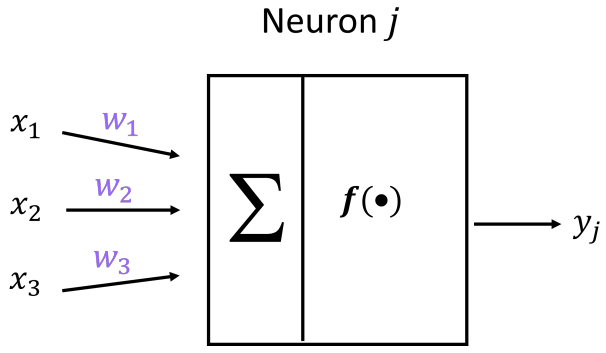


Figure 4.1. Computing the output of an artificial neuron, $y_j = f(\sum_i^n x_i w_i)$.

Meanwhile, Figure 4.2 shows the connection to compute a capsule and the steps to follow are:

1. Multiply the input vectors ($\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_n$) by the weight matrices ($W_{1j}, W_{2j}, \ldots W_{nj}$), which encode spatial relationships between low-level features and high-level features (matrix multiplication).

2. Multiply the result ($\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$) by the coupling coefficients ($c_1, c_2, \ldots c_n$) .

3. Compute the weighted sum of the input vectors, i.e. $s_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$.

4. Apply an activation function (*squash*) to vector values (s_j) to get the output $\mathbf{v}_j$.

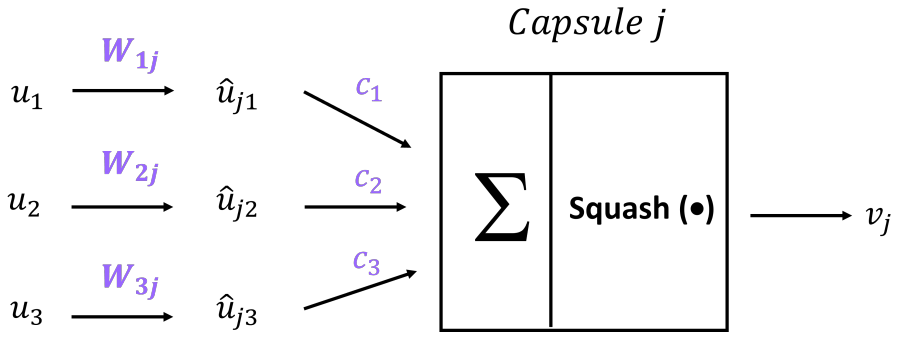


Figure 4.2. Computing the output of a capsule, $\mathbf{v}_j$.

Finally, Table 4.1 summarizes the differences between how an artificial neuron is computed and a capsule. In this table, the complexity of the internal operation behind the computed capsule can be appreciated.

Table 4.1. Important differences between capsules and neurons adapted from [9].

Differences between capsules and neurons			
		Capsule	Artificial Neuron
Input from low-level capsule/neuron		vector ($\mathbf{u}_i$)	scalar (x_i)
Operation	Affine Transform	$\hat{\mathbf{u}}_{j\|i} = \mathbf{W}_{ij}\mathbf{u}_i$	-
	Weighting Sum	$s_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j\|i}$	$z_j = \sum_i w_i x_i + b$
	Nonlinear Activation	$v_j = \frac{\|s_j\|^2}{1+\|s_j\|^2}\frac{s_j}{\|s_j\|}$	$y_j = f(z_j)$
Output		vector($\mathbf{v}_j$)	scalar (y_j)

4.2 Dynamic Routing by Agreement Algorithm

Figure 4.2 shows the connections and parameters needed to compute a capsule in general terms. Next, this section describes how to obtain all the necessary elements to carry out the procedure of the routing by agreement algorithm to obtain the output vectors of a Capsule network.

In the dynamic routing algorithm, the output obtained from the previous step is multiplied by the weights of the network. In a usual ANN, the weights are adjusted based on the error rate, followed by backpropagation [62]. However, this mechanism is not applied in a Capsule Network. Dynamic routing is what determines the modification of weights in a network. A Capsule Network adjusts weights so that a low-level capsule is strongly associated with high-level capsules that are in its proximity.

The capsule predictions are made by multiplying each capsule by a weight matrix $\mathbf{W}_{ij}$ where i is the number of the capsule and j is the number of the total classes that the algorithm is trying to predict (i.e. the MNIST dataset has ten classes). The matrix $\mathbf{W}_{ij}$ is very important because it captures the spatial relationships between the lower level features and the higher level features performing the affine transformation. Each weight is actually a matrix 8×16, so each prediction is a matrix multiplication between the capsule vector and this weight matrix, as can be seen in Figure 4.3. Therefore, each prediction is a 16-degree vector. It is important to note that the 16 dimension is an arbitrary choice, just as 8 is the size of the capsules.

As mentioned above, each entry of an output vector in a capsule represents the probability that the associated entity is present in the current input. Therefore, it is necessary to use the non-linear squashing function because only the length of the vector changes, not the orientation. Also, the squashing function obtains a vector with values 0 and 1; ensuring that small vectors take values close to 0 and large vectors get values below 1. Equation 4.1 shows the squashing function proposed by Sabour *el at.* [7], where v_j is the vector output of the capsule j and s_j is its total input.

$$v_j = squash(s_j) = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|} \tag{4.1}$$

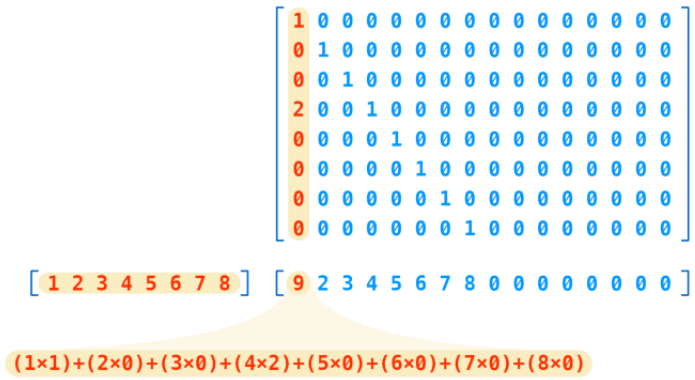


Figure 4.3. Example of the multiplication between a capsule and the weight matrix $\mathbf{W}_{ij}$[6].

According to Sabour *et al.* [7], for all but the first layer of capsules, the total input to a capsule s_j is a weighted sum of all prediction vectors $\hat{\mathbf{u}}_{j|i}$ of the capsules in the layer below by multiplying the output $\mathbf{u}_i$ of a capsule in the layer below by a weight matrix $\mathbf{W}_{ij}$, which means $\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$ and $s_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$. This operation uses a weight transform matrix as mentioned above, which encodes the spatial importance and other relations between the characteristics of the low-level capsules and the current one. If one of the calculated prediction vectors has a high value with a possible parent, then there is downward feedback where the values of the coupling coefficients (c_{ij}) are adjusted to select the correct connection path by the iterative dynamic routing process, as explained in Algorithm 1. This results in a more intelligent selection than just choosing the most significant number, like in max-pooling.

The coupling coefficients between the capsule i and all the capsules in the layer above sum up to 1 and are determined by a softmax routing function, whose initial logits b_{ij} are the log prior probabilities that the capsule i should be coupled to the capsule j as shown in Equation 4.2.

$$c_{ij} = softmax(b_i) = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})} \tag{4.2}$$

We detail the complete dynamic routing by agreement process in Algorithm 1 , where the parameter r is the selected iteration number and the value l is the number of the current layer. The main ideas of algorithms lines are Line 1: This line defines the procedure of ROUTING, which takes affine transformed input $\hat{\mathbf{u}}_{j|i}$, the number of routing iterations r, and the layer number l as inputs. Line 2: b_{ij} is a temporary value that is used to initialize c_i in the end. Line 3: The for loop iterates r times. Line 4: The softmax function applied to b_i makes sure to output a non-negative c_i, where all the outputs sum to 1. Line 5: For every capsule in the succeeding layer, the weighted sum is computed. Line 6: For every capsule in the succeeding layer, the weighted sum is squashed. Line 7: The weights b_{ij} are updated here. Where $\hat{\mathbf{u}}_{j|i}$ denotes the input to the capsule from low-level capsule i, and v_j denotes the output of high-level capsule j.

Algorithm 1 Routing Algorithm, according to Sabour et al. [7]

1: **procedure** ROUTING ($\hat{\boldsymbol{u}}_{j|i}, r, l$)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l : $c_{ij} \leftarrow softmax(b_i)$ ▷ $softmax$ computes Eq.4.2
5: for all capsule j in layer $(l+1)$: $s_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule i in layer $(l+1)$: $v_j \leftarrow squash(s_j)$ ▷ $squash$ computes Eq.4.1
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij}$ + $\hat{\mathbf{u}}_{j|i}\, v_j$
8: **return** v_j

To summarize, the algorithm starts by calculating the mean of all the predictions, and each prediction starts out with equal importance. Then they measure the distance between every point from the mean. The further the point is from the mean it is less important and disappears. After that, the algorithm recalculates the mean, this time taking into account the importance of the point. In the paper, they do this cycle 3 times. The highest agreeing points end up getting passed on to the next layer with the highest activation.

4.2.1 Margin Loss for Digit Existence

The margin loss function, Equation 4.3, was proposed by Sabour in the original CapsNet paper. According to Sabour, the margin loss represents the probability that a capsule entity exists based on the length of the instantiation vector [7]. To allow multiple digits, we use a separate margin loss, L_k for each digit capsule k

$$L_k = T_k max(0, m^+ - \parallel \mathbf{v}_k \parallel)^2 + \epsilon(1 - T_k)max(0, \parallel \mathbf{v}_k \parallel - m^-)^2 \tag{4.3}$$

In Equation 4.3, $\mathbf{v}_k$ is the vector obtained from DigitCaps layer, T_k is equal to one if a digit of class k is present and m^+= 0.9 and m^-= 0.1. We use ϵ = 0.5. So, the first term of the equation represents the loss for a correct DigitCaps, and the second term represents the loss for an incorrect DigitCaps. The total margin loss is the sum of the losses of all class capsules.

4.3 CapsNet Original Architecture

The original CapsNets architecture is shown in Fig. 4.4, and this architecture has only three layers: Convolutional Layer (ConvLayer), PrimaryCaps layer, and DigitCaps layer [7]. Moreover, CapsNets have a Reconstruction stage formed by three FC layers, as shown in Figure 4.5. According to Sabourn [7], the dataset used to train this architecture was MNIST, because of that, the network has ten classes and handles images of size 28 $\times$ 28 as we can see in Figure 4.4.

Convolutional Layer (ConvLayer)

It is used to extract the main features of the input image. The original Sabour architecture selects 256 channels, or filters, with a kernel of 9×9 parameters, with a stride of 1 and the ReLU function as shown in Figure 4.4. This layer converts the intensity of pixels into local feature activity detectors and uses them as input for the PrimaryCaps Layer [7].

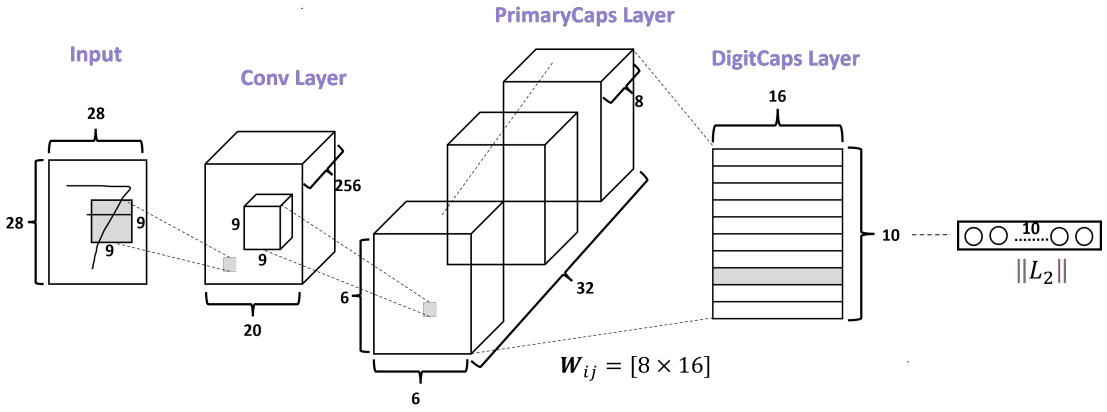


Figure 4.4. A simple CapsNet architecture with 3 layers, adapted from [7].

PrimaryCaps Layer

Given the 256 channels with the size of the featured maps of 20×20 from the ConvLayer, as shown in Figure 4.4, a kernel of 9×9 parameters with a depth of 256 filters and a stride of 2 is applied to form the Primary Caps layer. This layer is made up of 32 Capslayers each of size $6 \times 6 \times 8$, where each Capslayer has 36 capsules with eight dimensions (8D). Therefore, this operation generates 1,152 capsules.

DigitCaps Layer

Once the capsules are computed, the network decides which information will be passed to the next layer. Capsule predictions are made by multiplying each capsule by a weight matrix $\mathbf{W}_{ij}$ for each class that we are trying to predict (the MNIST dataset has ten classes). Each weight is actually a 8×16 matrix, so each prediction is a matrix multiplication between the capsule vector and this weight matrix, as can be seen in Figure 4.3. Therefore, we will end up with 11,520 predictions and each prediction is a 16-degree vector. It is important to note that the 16 dimension is an arbitrary choice, just as the 8 is the size of the capsules.

The next step is to find out which of these 11,520 predictions agree the most with each other

using the dynamic routing algorithm explained in Section 4.2. Finally, the DigitCaps layer ends with ten 16-dimensional vectors, as shown in Figure 4.4, one vector for each digit. This layer is the final prediction, and it can produce two outputs. The first output consists of ten vectors produced by the DigitCaps layer, where each vector corresponds to each class in the network. This output then uses the norm L_2 to calculate the length of each vector. Finally, the vector values are the confidence in the detection of the associated class. Therefore, the vector with the highest value is the prediction. The second output is the reconstruction stage, which we will explain in the next section.

Reconstruction Stage

Finally, there is a reconstruction stage where the model uses only the activity vector of the correct digit capsule of the DigitCaps Layer as input for recreating the original input image. The reconstruction stage on the Sabours architecture is formed by two fully connected layers with 512 and 1024 parameters, as shown in Figure 4.5. It is also important to note that, in the reconstruction stage, specifically, the decoder is part of the network that could generate more parameters due to its fully connected layers. In the architecture shown in Figure 4.4, the reconstruction stage generates 1,411,344 parameters. For these reasons, some models prefer to leave this stage out of their architectures, mainly if they handle complex RGB images.

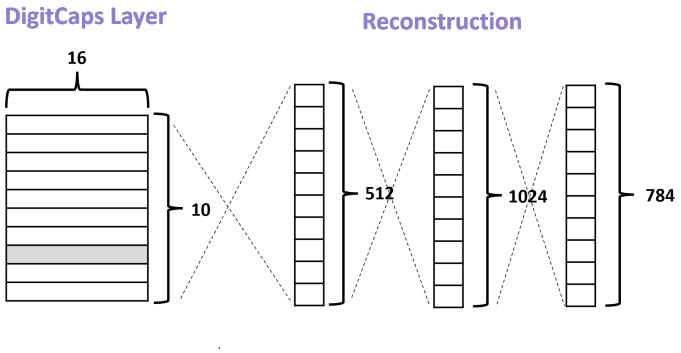


Figure 4.5. The Reconstruction Stage [7].

Then, the model minimizes the distance between the reconstructed and original images by a loss function called reconstruction loss. Also, the decoder scales down the reconstruction loss by

0.0005 so that it does not dominate the margin loss during training. The original architecture uses the mean square error (*mse*) as a reconstruction loss. Therefore, the reconstruction loss promotes the correct reconstructions of the input data. These functions are used to encourage the capsules to encode the instantiation parameters of the input class; so, this loss acts as a regularizer. In the following, we present the details of the reconstruction losses used in this thesis.

Reconstruction Losses

mae loss

The mean absolute error (*mae*) takes the difference between the model prediction and the ground truth, applies the absolute value to that difference, and then averages it across the entire dataset. Therefore, all errors will be weighted on the same linear scale.

$$mae = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i| \tag{4.4}$$

mse loss

The mean squared error (*mse*) takes the difference between the model prediction and the ground truth, squares it, and averages it across the whole dataset. This function ensures that the training model does not have outlier predictions with huge errors.

$$mse = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2 \tag{4.5}$$

categorical cross-entropy loss

The categorical cross-entropy loss (*cc*) is a loss function for multi-class classification model which classifies the data by predicting the probability of whether the data belongs to one class or the

other class. This function penalizes the model when it estimates a low probability for the model prediction.

$$cc = -\sum_{i=1}^{n} x_i \cdot log(y_i) \tag{4.6}$$

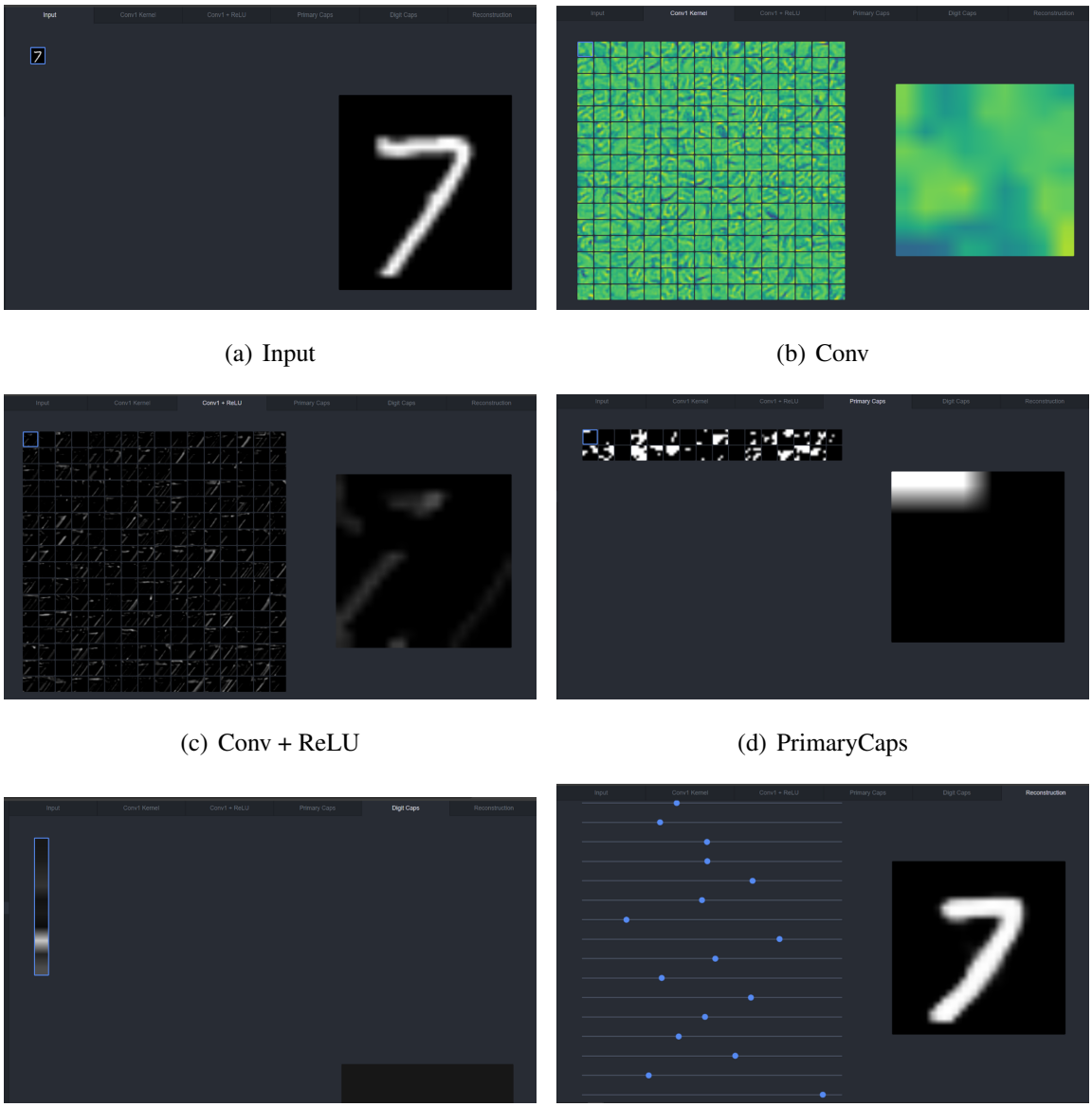
What Happens Inside a CapsNet

The layers described in Section 4.3 are shown graphically in Figure 4.6. In this figure, we can see how the information of a single input image passes through the network. Figure 4.6(a) shows the image using the network as input; in this case, is the digit 7. Next, Figure 4.6(b) shows the 256 feature maps obtained in ConvLayer. Figure 4.6(c) shows the same maps, but adds in each map the non-linear function ReLU. Now, Figure 4.6(d) shows the 32 Capsule Layers generated in the Primary Capsule Layer. In Figure 4.6(d) the first capsule layer is selected, this layer is made up of 36 capsules represented as a vector. In this case, the white color represents a high possibility of presence of an important feature. Next, Figure 4.6(e) shows how the Digit Capsule Layer is formed. In this layer, each position is a 16-dimensional vector where the magnitude is calculated. A low magnitude is represented by a black color, while a high value is represented by a white color. It can be seen that the white color agrees with the position of the number seven. Finally, figure 4.6(f) shows the digit reconstructed from the Digit Caps layer.

4.4 CapsNets Advantages

The CapsNet architecture has several advantages over the use of CNN. First, note that, whereas a traditional CNN considers only a good performance when the model predicts the correct digit. CapsNet uses the reconstruction stage to improve the results because it considers not only the correct digit, but also a correct reconstruction. An example is shown in Figure 4.7.

In addition, the CapsNet Reconstruction Stage allows the network to generate additional real data by simply adjusting the instantiation parameter values encoded in the Digit Layer. An example of this idea is shown in Figure 4.8, where all digits are generated by slightly changing the value



(a) Input (b) Conv

(c) Conv + ReLU (d) PrimaryCaps

(e) DigitCaps (f) Reconstruction

Figure 4.6. Example of the Capsnets internal layers.

of the instantiation parameter of the output vector. This advantage is already used to generate new labeled data from existing data in a more realistic way than in the data augmentation process [89]. For example, the TextCaps architecture achieves state-of-the-art results on the EMNIST-letter dataset with only 200 images per class when originally the dataset has 5,600 images per class [86, 163].

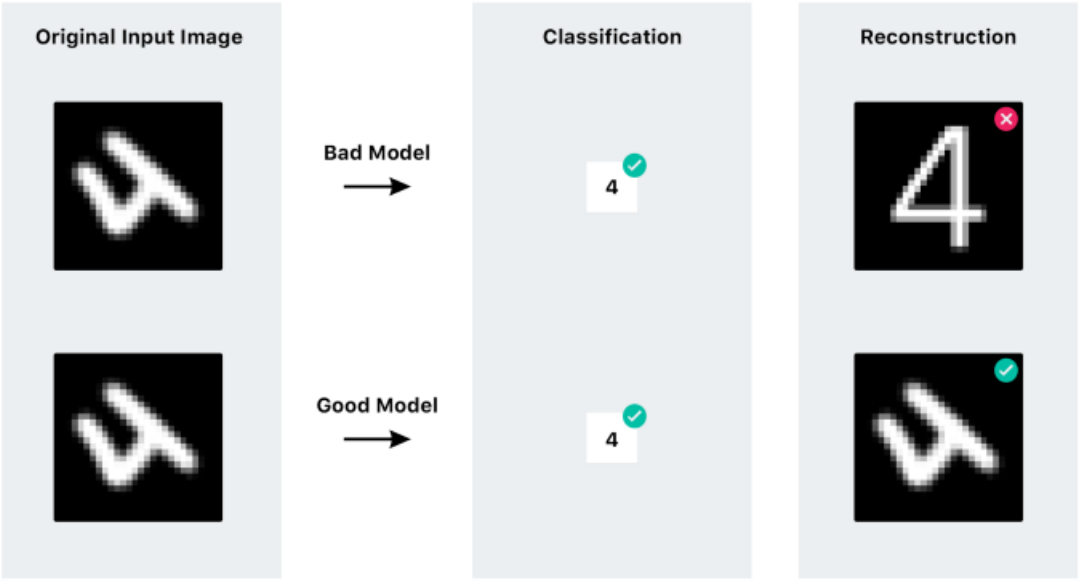
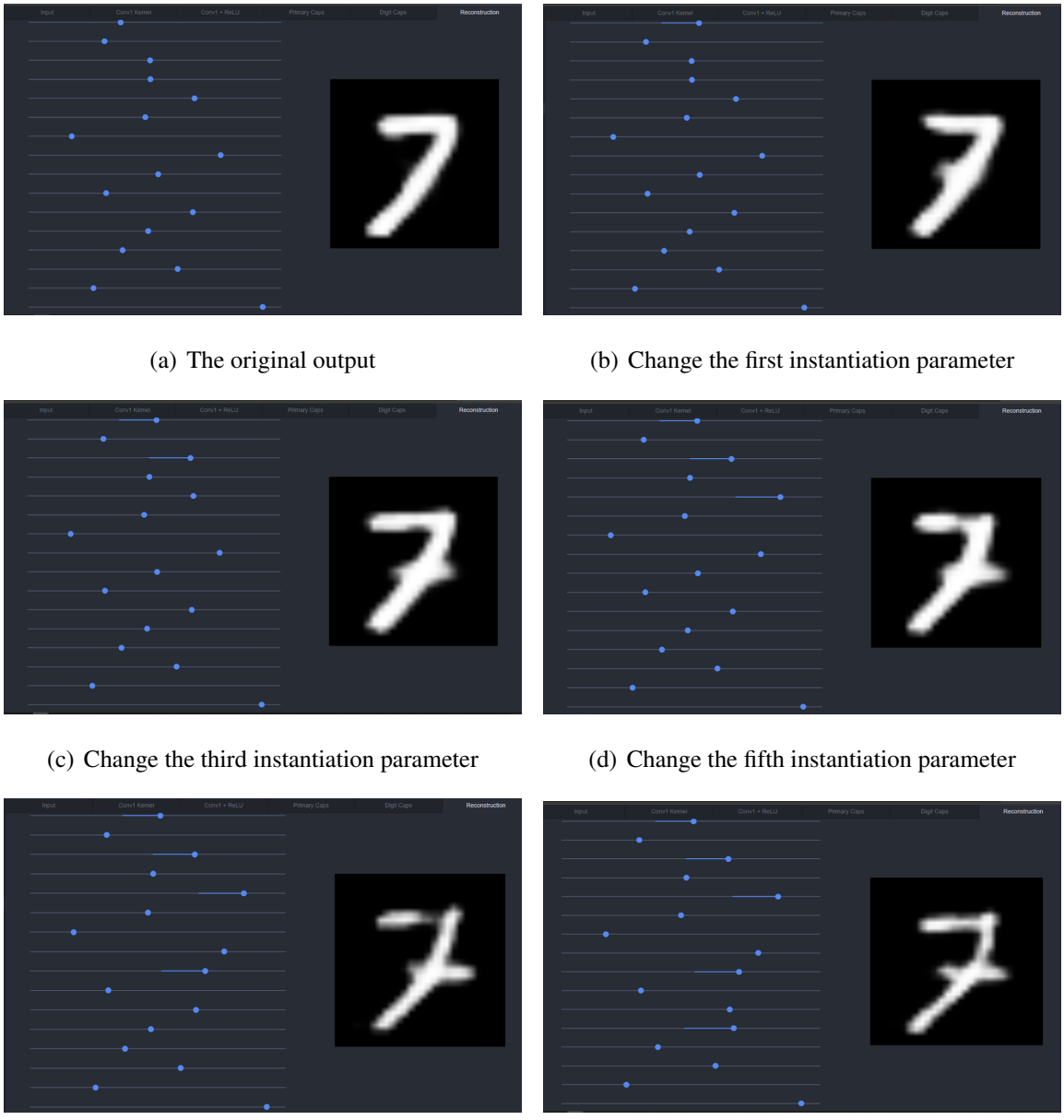


Figure 4.7. Example of reconstruction stage in CapsNet [6].

Another great advantage of the CapsNet architecture is the Robustness to Affine Transformations [7]. In the original CapsNet paper, the researchers designed a CNN, called baseline, to achieve the best performance on MNIST while keeping the computation cost as close to CapsNet to compare results. For this experiment, they used two MNIST-derived datasets called Expanded MNIST (for training) and affMNIST [1] (for the test). In the first dataset, each example is an MNIST digit randomly placed on a black background of 40×40 pixels. The second dataset consists of a small random affine transformation of each digit. The precision results are shown in Table 4.2. It can be seen that the two architecture presents a similar performance on the training. However, in the testing CapsNets overcomes the CNN network.

Another great result of CapsNet is the ability to segment highly overlapping digits [7]. For this experiment, they also developed a new dataset called MultiMNIST where they overlapping one digit on top of another in a percentage of 80. In this dataset for each digit of the MNIST, a thousand examples of the MultiMNIST were generated. It is very important to highlight that this network

[1] https://www.cs.toronto.edu/ tijmen/affNIST/



(a) The original output

(b) Change the first instantiation parameter

(c) Change the third instantiation parameter

(d) Change the fifth instantiation parameter

(e) Change the ninth instantiation parameter

(f) Change the twelfth instantiation parameter

Figure 4.8. Example of how a CapsNet can generate new real data.

was trained from scratch with 60M of images for training and 10M for testing. The images of the results of this experiment are shown in Figure 4.9. In the image **L**: (l_1, l_2) represents the label of the two digits in the image and **R**: (r_1, r_2) represents the two digits used for the reconstruction.

Another advantage using CapsNets is their adversarial robustness. Hinton compares a

Table 4.2. Results of affine Transformation.

Architecture	Expanded MNIST	affMNIST
Baseline	99.22%	66%
CapsNet	99.23%	79%

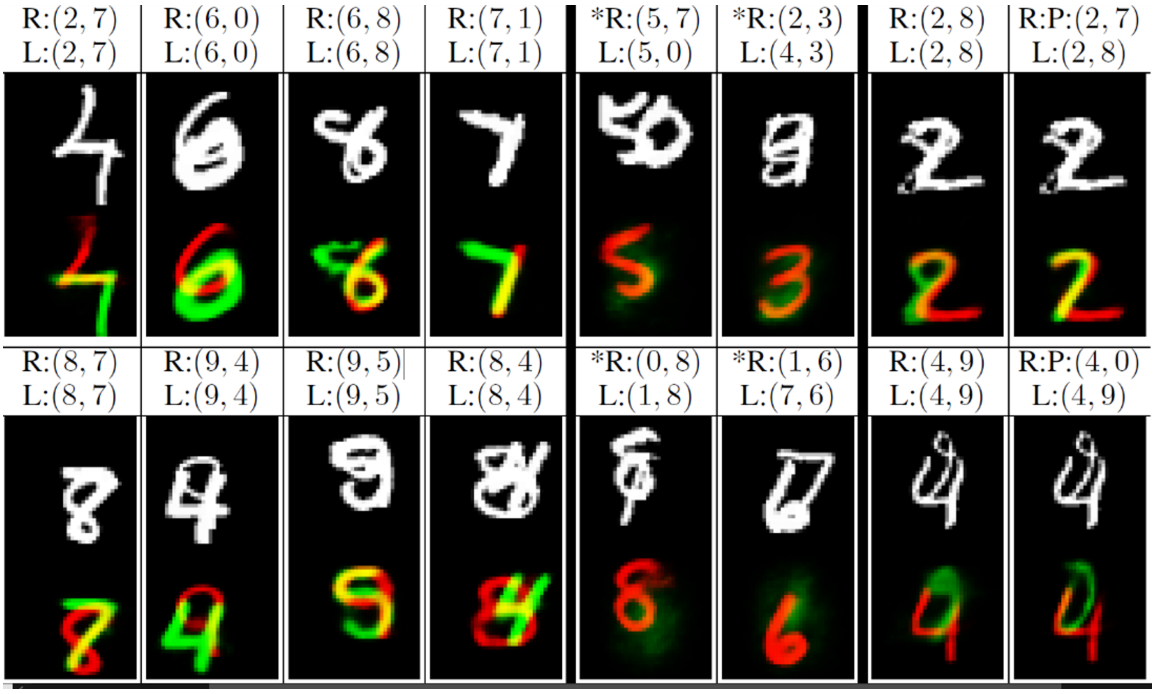


Figure 4.9. Sample reconstructions of a CapsNet on MultiMNIST test dataset [7].

Capsule model and a traditional CNN on the ability to deal with adversarial attacks [8]. In the paper, they generated two adversarial attacks using FGSM and BIM methods. They noted that the accuracy of the capsule model after the untargeted attack using FGSM never drops below chance (20%), while the precision of the convolutional model is reduced to significantly below chance with a ϵ as small as 0.2 as shown Figure 4.10. With respect to the BIM method the CapsNet model is much more robust to attack than the traditional convolutional model as shown Figure 4.10

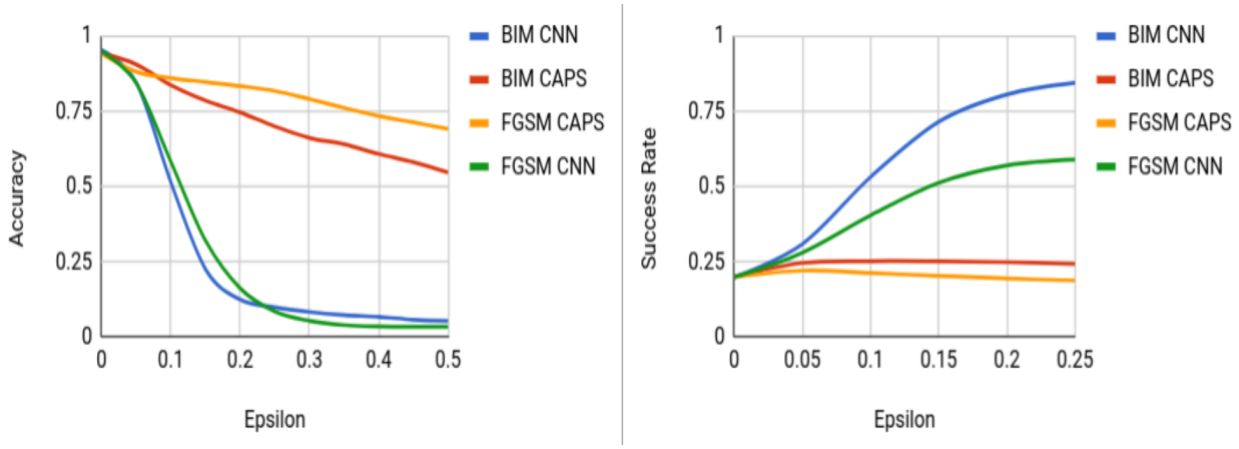


Figure 4.10. CNNs vs CapsNet under adversarial attacks [8].

4.5 CapsNets Limitations

Since the publication of Sabour et al.[7], some researchers have promoted the development of CapsNet. However, CapsNets presents some limitations, and some researchers are studying these shortcomings at each stage. Table 4.3 shows some limitations reported and the ongoing improvements suggested for CapsNets at each stage.

The main limitation of CapsNets is that this architecture has only been successfully tested in simple datasets. These datasets are grayscale images with a low resolution (28×28 or 32×32) such as MNIST, Fashion MNIST [38], EMNIST-letter [86], EMNIST-balanced [52], EMNIST-digit [163]. So, CapsNets are only tested successfully in some variation of the MNIST dataset, but they struggle on a more complex image. According to Table 4.3 some reasons for these limitations are:

- The architecture struggles to understand the entire context of the image
- CapsNet architecture is not suitable for complex background images due to its weak ability to extract features in the convolutional layer.
- They generate a large number of training parameters which translates into a great computational effort, especially in the routing by agreement algorithm where the algorithm

Table 4.3. Some CapsNets limitations reported.

Stage	Limitation	Improvement
Input	Only been tested on basics datasets.	• Adds new layers to adjust the data dimension [89],[117]
ConvLayer	The first convolutional layer performs a poor feature extraction.	• Increase or change number of convolution layers before capsule layer [104]. • Change how primary capsules are created [115].
PrimaryCaps Layer	The shallow depth of architecture does not allow us to understand more complex images.	• Stacking more capsule layers [8]. • Increasing the number of primary capsules [104]. • Change the way in which the capsule layers are formed [115],[95].
DigitCaps Layer	The computational cost of routing by the agreement algorithm is high [7],[95].	• Propose a new activation function [87]. • A new way of routing by agreement [102]. • Change DigitCaps Layer [90]. • Allows for parallel processing [8].
Reconstruction	The regularization technique works well only in two dimensional images.	• Generate new labeled data [89]. • Modify the scale of the reconstruction loss [8]. • Use none of the above categories [86]. • Replaces the FC with a deconvolutional network [86].

has an extra cycle of iterations (is the number of neurons inside the capsule) instead of a normal iteration to get the output.

- CapsNet tends to explain everything in the image, which is not suitable for image classification tasks with complex backgrounds.

For these reasons, some researchers have focused on combining the advantages of CNNs with the advantages of CapsNets as shown in Table 4.4. For example, Yang *et al.* propose the RS-CapsNet network [39], which uses some ideas from the ResNet architecture and the Squeeze and Excitation block, both of which were ILSVRC winners [39, 12]. The experimental results show that RS-CapsNet performs better on the CIFAR10, CIFAR100, SVHN, FashionMNIST and AffNIST datasets. It can also provide better translation equivariance and the number of training parameters is reduced by 65.11% compared to the CapsNets original architecture. So far, the preliminary results of the CapsNets approach are optimistic; however, there is still work to be done in order to achieve state-of-the-art results on complex datasets.

Table 4.4. Examples combining CNN with CapsNets.

Network	Dataset	Combination
DA-CapsNet [40]	MNIST, CIFAR10, FashionMNIST, SVHN, smallNORB and COIL-20	Attention layers + CapsNet
FSSCaps-DetCountNet [41]	The aerial elephant and The livestock	FSS classifier + CapsNet
DE-CapsNet [42]	CIFAR-10, Fashion MNIST	SGE + CapsNet + DCNet++
RS-CapsNet [39]	CIFAR10, CIFAR100, SVHN, FashionMNIST, and AffNIST	Res2Net + SE + CapsNet
ResCapsNet [43]	LiDAR	ResNet + CapsNet

4.6 CapsNets and CNNs in the Medical Field

Recently, some papers have also used CapsNets combined with CNNs for medical diagnostics. Mobiny and Van Nguyen [25] used chest CT scans for the diagnosis of lung cancer, using images of 32 $\times$ 32 pixels, and considered two classes: nodule and non-nodule obtaining. Their network achieves an accuracy of 88.55% with only 226 images and struggles with the reconstruction stage; for that reason, they add a convolutional decoder. Afshar et al. [26] used CapsNets to diagnose the type of brain tumor; they trained with 3064 MRI images with a small resolution (64$\times$64 pixels) and obtained a classification accuracy of 78% with three tumor classes: Meningioma, Pituitary and

Glioma. Kruthika et al. [28] proposed a CBIR system that uses the 3D capsule network, the 3D convolutional neural network, and pre-trained 3D autoencoder technology for early detection of Alzheimer's. They used MRI images with a size of 64 $\times$ 64 pixels for Alzheimer's diagnosis with a classification accuracy of 94.06%. Xiang et al. [164] used a combination of CapsNet and ResNet for automated breast ultrasound tumor diagnosis. Their dataset contains 444 images of 128 $\times$ 128 pixels. They managed two classes (malignant or benign), achieving and obtaining an accuracy of 84.9%.

Respiratory diseases can also be detected by analyzing radiological images. Mittal et al. [165] used convolutions and dynamic capsule routing to diagnose pneumonia on 5,857 chest radiographs with 100 $\times$ 100 resolution and obtained an accuracy of 95.9% to classify normal or pneumonia. Khanna et al. [166] developed the Detail Oriented Capsule Networks (DECAPS) model for the automatic diagnosis of COVID-19 using 746 chest CT images with a size of 448 $\times$ 448 pixels. In addition, they used GANs for data augmentation. Their model achieved 87.6% accuracy in detecting two classes: Patients with COVID-19 and non-COVID-19. Afshar et al. [167] achieved a detection accuracy of 95.7% in two classes using their COVID-CAPS model. They used training images of 224 $\times$ 224 pixels and a transfer learning approach tuned with a new dataset constructed from an external dataset of X-ray images. Toraman et al. [168] proposed a convolutional CapsNet approach to detect COVID-19 disease from X-ray images using capsule networks. They used CT images with a size of 128 $\times$ 128 to detect three classes: COVID-19, no findings, and pneumonia. In addition, they used the max-pooling operation and data augmentation. Their model achieved an accuracy of 84.22%.

4.7 DRCapsNet Models

To design a computational model it is necessary to configure a large number of hyperparameters. It is important to note that altering a hyperparameter may or may not have a considerable effect on the model's performance, and it is impossible to predict the outcome due to the complexity of the hyperparameters. Consequently, the only way to test the computational model performance it is to

retrain the model for each configuration, which is a complex task.

In order to design the DRCapsNet computational model, it is necessary to know that there are two types of hyperparameters, those that correspond to the architecture of the model and those for training. Figure 4.11 describes the CapsNet model training process. First, the input image is passed through the model architecture and generates two outputs, as shown in Figure 4.19. The model then uses a total loss function formed by two different functions: margin loss and reconstruction loss selected as explained in Section 4.2.1. The first is the function obtained from the model prediction and the actual target. The model prediction corresponds to the vector with the highest magnitude in the ClassCap layer. The second loss compares the image reconstruction (decoder output) and the original input. Reconstruction loss is multiplied by a λ hyperparameter that weights the relative contribution of each term. Therefore, it is summed with the margin loss to obtain the total loss. The λ hyperparameter is used on the original Sabours architecture and is used to avoid overfitting in the network; this value is very important for the design of the new model as will be described in the following sections. So, the reconstruction loss and the λ value are the hyperparameters used for improve the computational model.

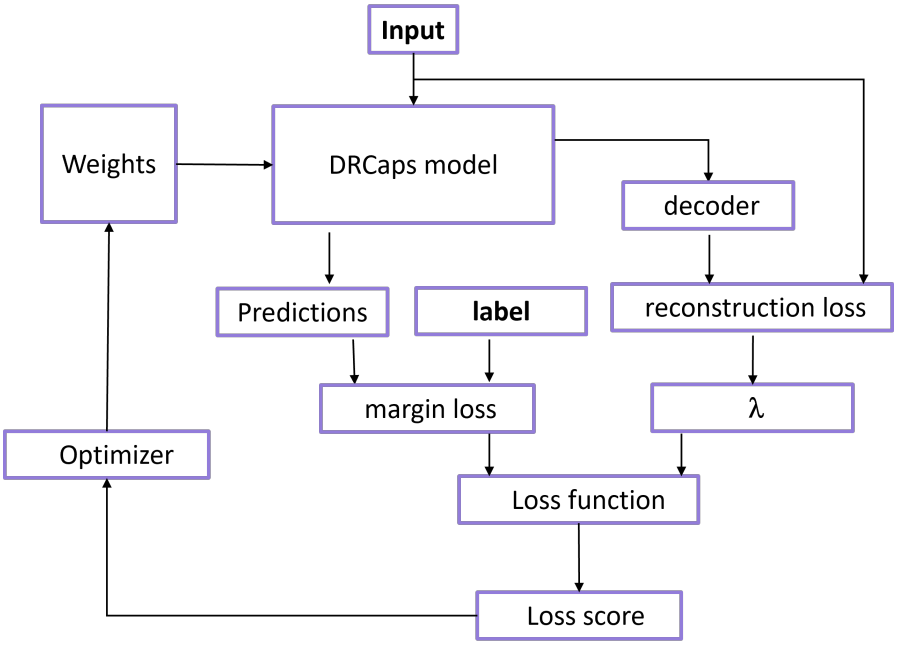


Figure 4.11. Training a CapsNet model.

On the other hand, Figure 4.12 shows at each stage in the CapsNet architecture the possible hyperparameters configurations to the architecture. As mentioned in Section 4.6, the combination of CapsNet and CNN could improve the performance of some models. Therefore, another contributions of this thesis is to propose a computational model based on CapsNets that can handle complex medical images. In order to achieve this goal, it was decided to focus on the ConvLayer stage because the original architecture performs a poor feature extraction. One contribution in the model is adding more convolutional layers at this stage, and instead of using the max pooling operation to reduce the size of the feature maps, it was decided to use the dilation rate and the stride hyperapameters in order to handle more complex input images and to control the number of capsules desired.

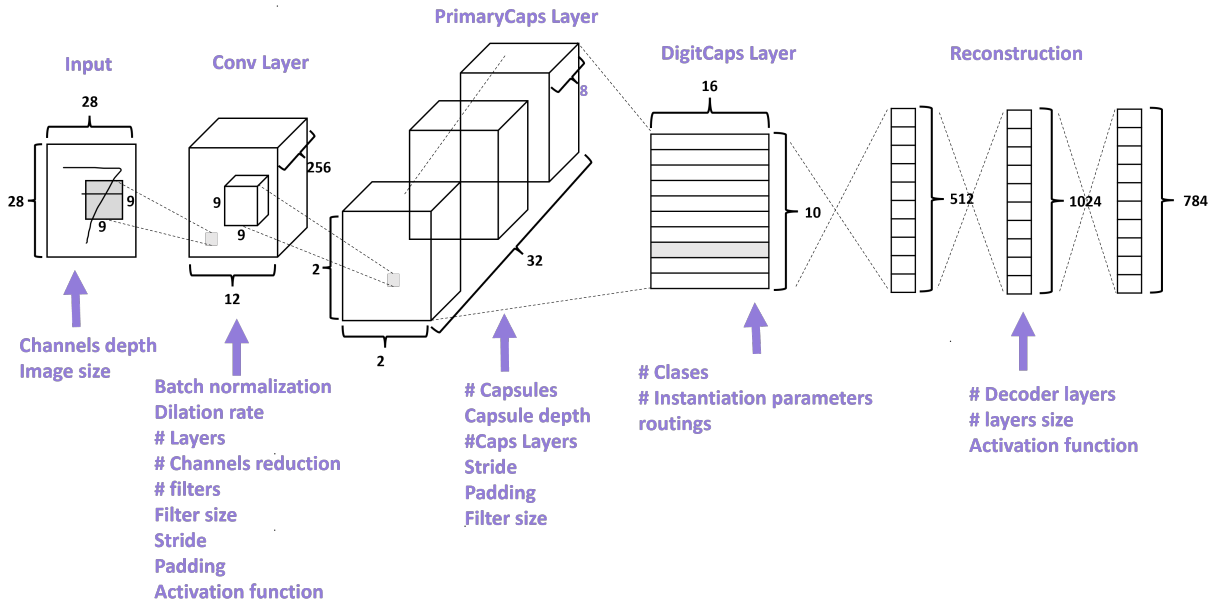


Figure 4.12. CapsNet possible hyperparameter configurations.

4.7.1 Dilation Rate

One of the contributions of this thesis work is the use of the hyperparameter dilation rate (DR) in some layers of the ConvLayer. The use of the dilation rate is particularly popular in real-time segmentation, where a wide field of view must be covered and multiple convolutions or larger kernels cannot be allowed [169, 170]. This hyperparameter modifies the convolutional kernel

by introducing gaps between elements. The number of spaces depends on the selected value and indicates the spacing between rows and columns of the kernel. Figure 4.13 illustrates what the dilation rate hyperparameter looks like with the values (2,2), (4,4), and (8,8) in a 3×3 kernel. For example, a dilation rate of (2,2) in a 3×3 kernel will cover the same receptive field as a 5×5 kernel using only nine parameters, as shown in Figure 4.13. Moreover, the same number of parameters of a 3×3 kernel can cover the same region of a 9×9 kernel or a 17×17 kernel with different dilation rates. Then, the new kernel shape is applied to the entire receptive field to obtain a new featured map. This procedure is known as a dilated convolution [170].

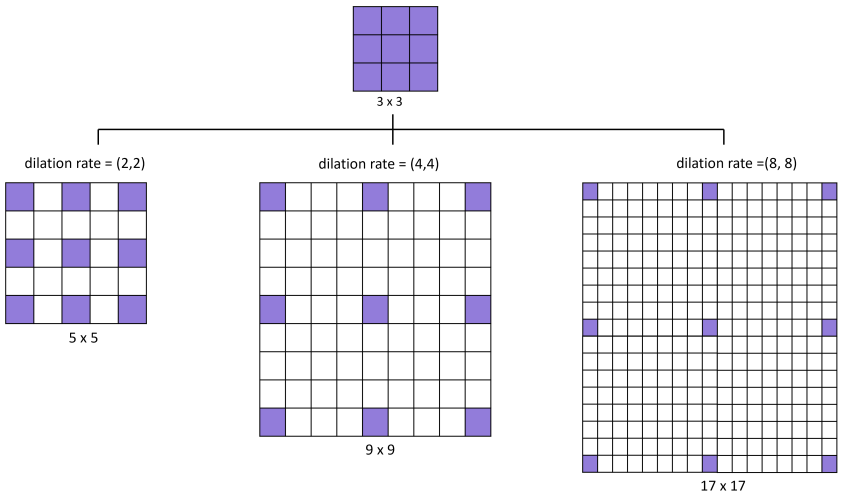


Figure 4.13. Examples of the space that covers different values of dilation rate.

One reason for using this type of layer is that it can cover the receptive fields with fewer parameters than with normal convolutional layers while generating the same size feature map at the output. This allows for the use of better quality images without increasing the complexity of the network. Figures 4.14 and 4.15 show the same receptive field, but we applied two different convolution operations. Figure 4.14 uses a 3x3 kernel with a stride equal to one and generates a 3x3 feature map using 9 weights and 81 operations as explained in Section 3.1.1. On the other hand, Figure 4.15 used a 2x2 kernel with a dilation rate equal to (2,2) and a stride equal to one, using only four weights and 36 operations. Both configurations generate a feature map with a size of 3x3 values.

It is important to mention that when using the hyperparameter dilation rate, it is necessary

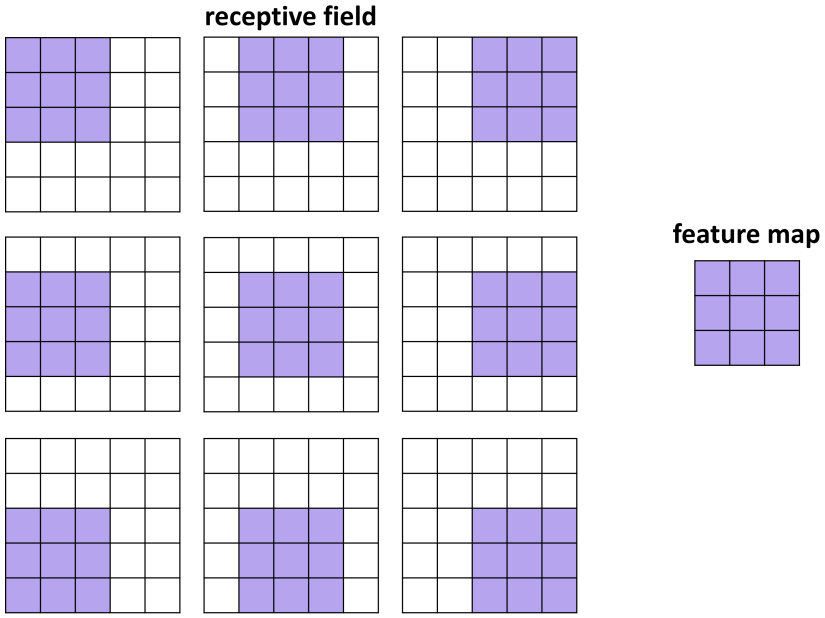


Figure 4.14. Convolutional operation with a stride = 1 and a dilated rate = (1,1).

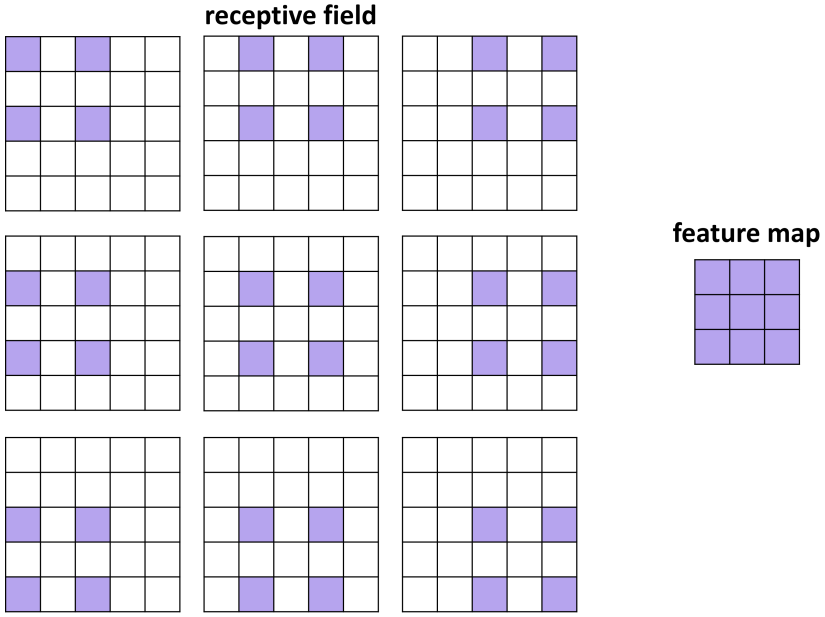


Figure 4.15. Convolutional operation with a stride = 1 and a dilated rate = (2,2).

to use a stride value equals to one. This in order to avoid the loss of information since the pixels that are omitted at the beginning with the extended kernel, are taken into account when the filter is displaced by one pixel. So it does not matter that the kernel passes only once through a pixel, the

capsules take care of giving significant importance to that piece of information. Therefore, some advantages using the dilation rate hyperparameter are:

- Exists less overlapping among pixels compared with a normal convolutional layer [170].
- The receptive field of units in the network can grow exponentially with the number of layers/parameters as compared to non-dilated convolutions [170].
- Makes the network training faster by performing fewer number of operations [170].

4.7.2 DRCapsNet Models on MNIST

Because one of the contributions of this thesis work is to see if the dilation rate hyperparameter can be a factor that improves the accuracy of the computational model; we started replicating the CapsNet baseline architecture in the MNIST dataset. For a better understanding, the number of parameters associated with each layers in the CapsNet original model is described at the bottom of the layers as shown in Figure 4.16.

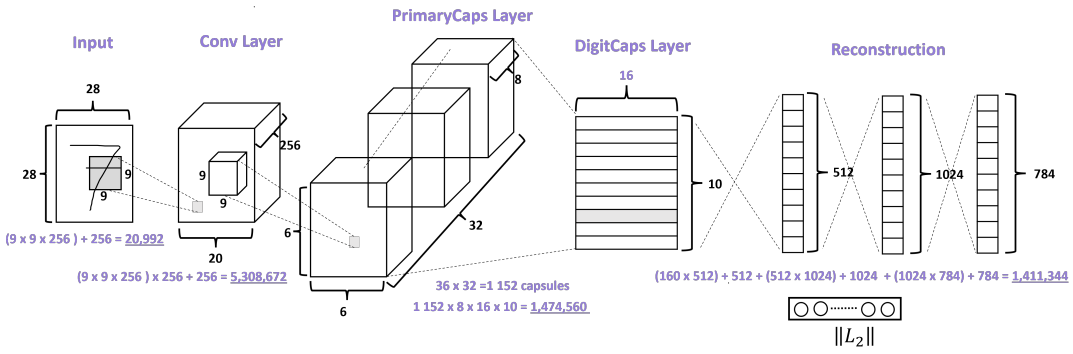
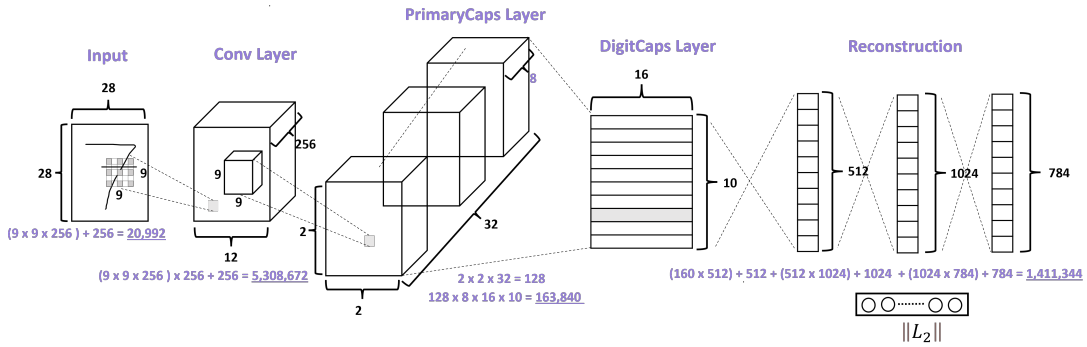


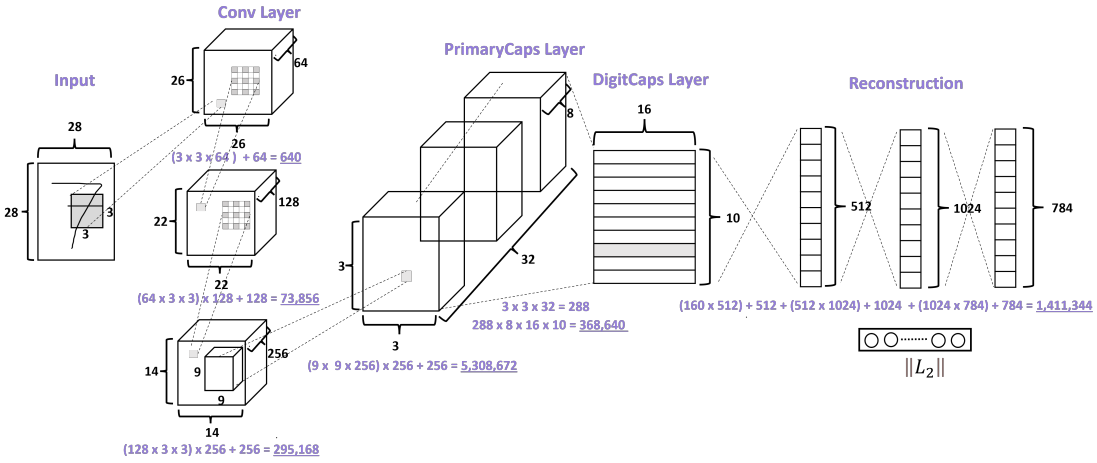
Figure 4.16. CapsNet original model.

Then, it was decided to probe on the same MNIST dataset, two different models (DRCapsNet V1 and V2) with different dilation rate values. For the first model (DRCapsNet V1) the smallest value of DR was selected to observe how this hyperparameter affects the original architecture. So, the DRCpasNet V1 uses a dilation rate of 2 which is implemented in the first ConvLayer as shown in Figure 4.17(a). Also, the DRCapsNet V1 maintains the same filter size in the input images, but

instead of using the stride hyperparameter, it uses a DR of (2,2), leading to a significant reduction in the number of capsules and the total parameters in the network. This is because the DR filter size behaves as an input filter of 19 $\times$ 19, reducing the size of the feature map to 12 $\times$ 12 values in the ConvLayer, thus causing the capsule reduction as seen in Figure 4.17(a).



(a) DRCapsNet V1 model.



(b) DRCapsNet V2 model.

Figure 4.17. The DRCapsNet models used on the MNIST dataset. For a better understanding, the number of parameters associated with each layers is shown at the bottom of the layers. The sum of these parameters can be seen in Table 4.5.

For DRCapsNet V2, it was decided to follow the CNN architecture idea, where the number of filters is increased in each layer while the feature size is reduced, as seen in Figure 3.1. To do this, the filter size was reduced from 9 $\times$ 9 to 3 $\times$ 3 parameters. The 3 $\times$ 3 size was selected

because by combining it with different DR values, the behavior of a larger filter can be achieved with fewer parameters as explained in Section 4.7.1. Then, two different DR values were added to simulate the passing of the network through different filter sizes, such as VGG or GoogLeNet. In Figure 4.17(b), it can be seen that the first two layers at the ConvLayer stage reduce the size of the feature maps in a small proportion, because these layers do not use the stride value instead it uses a DR value of (1,1) and (2,2) respectively. However, the last layer uses a DR of (4,4) simulating a 9×9 kernel, reducing the size of the feature map size to 14×14 and generating 288 capsules, as shown in Table 4.5.

Table 4.5. Summary of models used in MNIST.

Network	Filter size	DR	Stride	Capsules	Parameters	Accuracy
CapsNet	9 x 9	no	2	1,152	8,215,568	98.53%
DRCapsNet V1	9 x 9	(2,2)	no	128	6,904,848	99.22 %
DRCapsNet V2	3 x 3	(1,1),(2,2),(4,4)	no	288	7,458,320	99.61%

Table 4.5 presents a summary of the different models describes above. The accuracy achieved in each version is CapsNet = 98.53% , DRCapsNetV1 = 99.22 % and DRCapsV2 = 99.61%. So, DRCapsNet V2 presents the highest accuracy in the MNIST dataset. Therefore, it can be seen that DRCapsNet V2 outperforms in accuracy the other CapsNet version on the MNIST dataset. With these models we can observe how the dilation rate hyperparameter reduces the capsule number and, therefore, the total parameters in a CapsNet architecture. It is important to note that the size of the featured maps in the last layer at the ConvLayer stage determines the number of capsules on the PrimaryCaps Layer as shown in Figure 4.17. Also, from Table 4.5 we can realize that with a small number of capsules the network can achieve a better accuracy.

4.7.3 DRCapsNet Models in COVIDx V7A

In the previous section, it explained how the dilation rate reduces the capsule number and the total parameters, while the accuracy is increased. So, it was decided to use DRCaspsNet V2 in the COVIDx V7A dataset. Table 4.6 outlines the results of the implementation of DRCaspsNet V2 on the COVIDx V7A dataset. The first row in Table 4.6 displays the results obtained on the MNIST

dataset. The following rows reveals the large number of capsules and parameters generated by the COVIDx V7A dataset. Because the COVIDx V7A dataset uses images with a depth of 3, the number of capsules and parameters increases significantly in the reconstruction stage because it has to construct the same input images with three channels. These increases in the number of capsules and parameters, as shown in Table 4.6, which generates great computational complexity in the dynamic routing by agreement algorithm. This results in computer memory saturation and makes it impossible to complete training. In order to decrease the number of parameters, it was decided to use a single depth for the input images, as seen in the third row of Table 4.6. Nevertheless, this only decreased the number of parameters in the decoder, while the number of capsules stayed the same, leading to the same memory error being thrown by the algorithm.

Table 4.6. DRCapsNet models results.

Version	Dataset	Input size	Depth	Capsules	Decoder	Parameters	Accuracy
DRCaspsNet V2	MNIST	28×28	1	288	1 411 344	7 458 320	99.61%
DRCaspsNet V2	COVIDx V7A	256×256	3	438,048	202 073 600	375 963 520	-
DRCaspsNet V2	COVIDx V7A	256×256	1	438,048	67 724 800	241 613 568	-
CapsNet V1	COVIDx V7A	256×256	3	512	25 390 400	28 612 288	80 %

To achieve the objective of improving the DRCaps model it is necessary to reduce the number of capsules and parameters in order to train the model successfully. After an analysis of the number of parameters used per layer (see Figure 4.17(b)), it was noted that the two sections that handle the greatest number of parameters are the ConvLayer and the Reconstruction. To reduce the number of parameters in the ConvLayer it was decided to remove the DR hyperparameter for the moment, because if only this hyperparameter is used, then the size of the feature map is reduced very little as can be seen in Figure 4.17(b) where the size of the feature maps only changes from 28 to 26 and from 26 to 22. On the other hand, if only the stride hyperparameter is used, the size of the feature maps is reduced from 22 to 14. For these reasons it was decided to only use convolutional layers with stride equal to two. Also, the number of feature maps generated in each layer maintains the CNNs structure of going from small to large values, and as the objective is to reduce the number of parameters we only use 2 sizes: 64 and 128 as shown in Figure 4.18. Regarding the Reconstruction stage, to reduce the number of parameters it was decided to change the size of the Fully Connected

layers of the original architecture to 3 layers of size 64, 128 and 128 respectively. Significantly reducing the number of parameters as seen in Table 4.6.

The new CapsNet model (called CapsNet V1) has four convolutional layers with a stride equal to 2 in each layer. Figure 4.18 shows in detail the changes made to the new architecture and how it was adapted to the COVIDx V7A dataset. For example, Figure 4.18 has the input images with a depth of 3, and the ClassCap Layer has 3 rows instead of the 10 classes used by the MNIST dataset. The first results of this architecture show an accuracy of 80%.

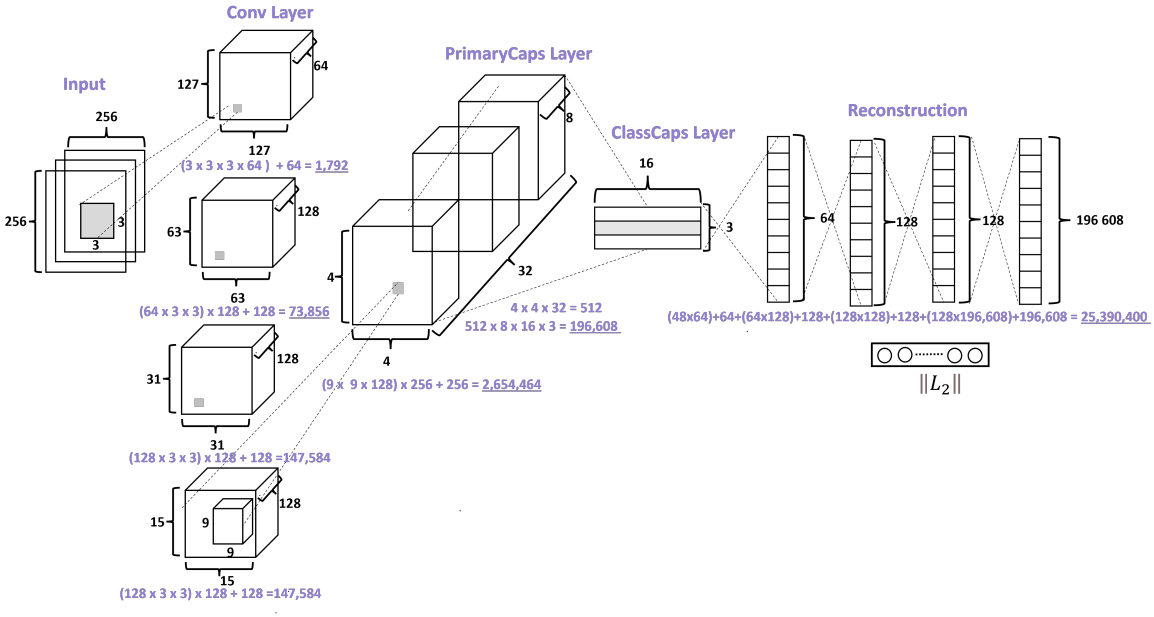


Figure 4.18. CapsNet V1 model.

So, in order to improve the accuracy it was decided to change only one hyperparameter at a time in the CapsNet V1 model to appreciate the cause of the change. It is important to highlight that there are no formula or equation that guarantees a successfully trained network while there are a lot of hyperparameters (model and training) which can change the model behavior, as shown Figure 4.12.

Before proposing different architectures to improve the performance of the CaspNet V1 computational model. Two experiments were done with two training hyperparameters: the reconstruction loss and the λ value as shown Figure 4.11. So, the first training hyperparameter

we decide to probe is the reconstruction loss at the Reconstruction Stage. It was decided to try three different reconstruction loss functions: *mse* (default), categorical cross-entropy (*cc*), and mean absolute error (*mae*) (see Section 4.3). As can be seen in Table 4.7, the reconstruction loss function that offers the best accuracy result is *mae*. So, hereafter the reconstruction loss used for the next experiments is the *mae* loss function.

Table 4.7. Different reconstruction loss in the CapsNet V1 model.

	cc	***mse***	***mae***
capsules	512	512	512
training accuracy	1	1	1
validation accuracy	0.7949	0.7832	0.7733
testing accuracy	0.7875	0.80	**0.825**

Despite the small improvement in accuracy, Table 4.7 shows that the model overfits because the training accuracy reaches 1 while testing and validation are lower. For these reasons, it was decided to change the training hyperparameter λ which works as a regularizator in order to avoid overfitting. The original CapsNet paper uses MNIST images with a resolution of 28×28, so they multiplied this resolution by 0.0005 and obtained a value of λ equal to 0.392. This value was used in the first experiments, but the COVIDx V7A used images with a higher resolution. Therefore, it was performed the same operation and obtained a value of λ equal to 32.768. This change increases the model accuracy and it reduces the overfitting in the network as shown Table 4.8.

Table 4.8. Different configuration of the CapsNets V1 model.

	CapsNet V1	**CapsNet V1**
weights	28,612,288	28,612,288
depth	3	3
λ	0.392	32.768
training acc	1	0.991
validation acc	0.7733	0.8025
testing acc	0.825	**0.85**

Next, it was decided to change the depth of the input image to 1 to reduce the number of parameters and try to improve accuracy. This modification correspond to the CapsNet V2 as shown in Table 4.9. It can be seen that the number of parameters decreases to 11, 702, 848 whereas the model with a depth equal to three the parameters are 28,612,288. In addition, Table 4.9 shows how the value of λ increases the accuracy of the model regardless of the depth of the input images. In addition, the value of λ only affects the accuracy of the model and no other hyperparameter such as the number of capsules or parameters.

Table 4.9. Accuracy (acc) results at different hyperparameters configurations.

Network	**Depth**	**DR**	**Stride**	**Capsules**	**λ**	**parameters**	**acc**
CapsNet V1	3	no	2	512	0.392	28,612,288	80.0
CapsNet V1	3	no	2	512	32.768	28,612,288	85.0
CapsNet V2	1	no	2	512	0.392	11,702,848	78.0
CapsNet V2	1	no	2	512	32.768	11,702,848	**86.2**

Now, we have a computational model with a sufficient number of parameters to be able to be trained with the COVIDx V7A dataset, it was decided to probe different model configurations in the ConvLayer Stage using CapsNet V1 as a backbone and start including the DR hyperparameter. Table 4.10 explains how each DRCapsNet models are formed. For the design of the models it is necessary to point out that if we use the DR value equal to (8,8), (4,4) and (2,2), these will behave like a filter of size 17×17, 9×9 and 5×5 respectively if we use a kernel size of 3×3.

The DRCaps V3 model reintroduces the DR hyperparameter with a value of (8,8) in the first convolutional layer. The idea in this experiment is to go through the input image (256×256) with a large filter (17×17) with as few parameters as possible. In addition, it was decided to use the max-pooling hyperparameter to test if it is really a necessary factor to reduce the size of the featured maps and increase the accuracy of the model. This model uses 11, 616, 832 parameters, in turn generates 288 capsules and obtains an accuracy of 81.22 % as shown in the third column of Table 4.10.

Table 4.10. Different architectures proposed for CapsNetV5. The parameter order indicates the number of filters, kernel size, stride and the dilation rate.

	CapsNet V2	**DRCapsNet V3**	**DRCapsNet V4**	**DRCapsNet V5**	**DRCapsNet V6**	**DRCapsNet V7**
conv1	64,3,2,(1,1)	64,3,1,(8,8)	64,3,1,(4,4)	16,3,1,(4,4)	128,3,1,(8,8)	128,3,1,(8,8)
conv2	128,3,2,(1,1)	128,3,1,(4,4)	64,3,2,(1,1)	16,3,2,(1,1)	128,3,1,(4,4)	64,3,2,(1,1)
maxpool		yes				
conv3	128,3,2,(1,1)	128,3,1,(2,2)	128,3,1,(4,4)	32,3,1,(4,4)	64,3,1,(2,2)	128,3,1,(4,4)
maxpool		yes				
conv4	128,3,2,(1,1)	128,3,2,(1,1)	128,3,2,(1,1)	32,3,2,(1,1)	64,3,2,(1,1)	64,3,2,(1,1)
maxpool		yes				
conv5			256,3,1,(4,4)	64,3,1,(4,4)	64,3,2,(1,1)	64,3,1,(2,2)
conv6			256,3,2,(1,1)	64,3,2,(1,1)	64,3,2,(1,1)	64,3,2,(1,1)
conv7			256,3,2,(1,1)	128,3,2,(1,1)	64,3,2,(1,1)	64,3,2,(1,1)
weights	11,702,848	11,616,832	15,574,272	11,331,376	11,371,712	10,192,128
capsules	512	288	128	128	3200	128
accuracy	86.25	81.22	82.14	82.58	77.00	88.00

DRCapsNet models V4 and V5 generate 128 capsules and obtain an accuracy almost similar to 82%. These models use the DR hyperparameter value equal to 4 in the first convolutional layer (conv1), to simulate a kernel of size 9×9 used in the original CapsNets architecture. The next layer of the models (conv 2) uses a stride equal to 2 and a DR equal to (1,1) to only reduce the size of the featured maps. This process is repeated two more times, and at the end another layer with a stride equal to 2 is added to reduce the size of the last featured maps and consequently the number of generated capsules. The only difference between the V4 and V5 models is the number of filters generated in each convolutional layer. The idea was to observe whether the model maintained its good performance with a smaller number of filters in each convolutional layer.

The principal idea in the DRCapsNet V6 model is that at first only the dilated layer is used and then the conv layers are used with a stride value equal to 2. Because, at first, the model only uses dilated layers, the featured size is reduced in a small proportion, which generates a large number of capsules (3200). One thing that caught our attention was that as we increased the number of capsules, the training time per period increased, while the accuracy decreased. For example, this model generates 3200 capsules and used a training time of 2 and a half minutes

per epoch, achieving an accuracy of 77%, while other configuration with only 288 capsules (DRCapsNet V3) achieved an accuracy of 81% as shown in Table 4.10.

Finally, the DRCapsNet v7 model shows the best accuracy and handles less weight than the other models. This model uses a combination of one dilated layer followed by a conv layer three times. Each time, the DR value decreases as the feature map size. In the end, another convolutional layer is added to reduce the number of capsules. To keep the model as simple as possible, only two filter sizes were used: 64 and 128. This model generates only 128 capsules and handles 10,192,128 parameters, achieving an accuracy of 88%, as shown in Table 4.10. Therefore, DRCapsNet V7A was selected as the final DRCapsNet model.

Finally, it was decided to conduct four more experiments with the λ hyperparameter in the new DRCapsNet model to improve network accuracy; Table 4.11 shows the results. The first experiment kept the parameters mentioned later. In the second experiment, we used $\lambda = 40$. In the third experiment (V3), we increase $\lambda = 80$, and the last experiment uses $\lambda = 0.328$. As shown in Table 4.11, the second experiment is the configuration that offers the best result. Therefore, we can observe that there is a relation between the input image size and the λ value.

Table 4.11. Accuracy (acc) results at different λ hyperparameter values.

Version	**Depth**	**DR**	**Stride**	**Capsules**	**λ**	**parameters**	**Acc.**
DRCapsNet	1	(8,8), (4,4), (2,2)	1,2	128	32.768	10,192,128	88.0
DRCapsNet	1	(8,8), (4,4), (2,2)	1,2	128	40	10,192,128	**90.0**
DRCapsNet	1	(8,8), (4,4), (2,2)	1,2	128	80	10,192,128	85.0
DRCapsNet	1	(8,8), (4,4), (2,2)	1,2	128	0.392	10,192,128	10.0

4.7.4 DRCaps Final Model

The DRCaps model is the main contribution of this thesis. Our model, unlike the original CapsNet model, increases the number of layers in ConvLayer to improve the extraction of features from the

input images. It also removes the max-poolig hyperparameter, instead of using dilated layers that reduce the number of parameters in the model. It also uses the stride hyperparameter to reduce the size of the feature maps and consequently the number of capsules of the original architecture. The number of classes in the ClassCaps layer is also modified to only 3 due to the selected COVIDx V7A data set. Finally, it reduces the size of the layers in the Reconstruction Stage to reduce the number of parameters when reconstructing the input image. The final architecture is shown in Figure 4.19. Note that the DRCaps model is sectioned into three stages, described below.

- **Convolutional Stage:** Aims to extract basic features from complex images using the dilation rate hyperparameter. The DR along with the stride allows the omission of a max-pooling operation and improves the spatial relationship problem.
- **Capsule Stage:** The Capsule Stage includes the Primary Caps and Class Caps because they are the only two layers that handle capsules. Primary Caps produces combinations based on the basic features detected by the Convolutional stage. The ClassCaps layer is the highest-level capsule layer that contains all the instantiation parameters. At this stage, the routing by agreement algorithm is implemented.
- **Reconstruction Stage:** Decodes the 16-dimensional vector from ClassCap into an image. It recreates the output image without the loss of pixels. They force capsules to learn the features that are useful for reconstructing the image. Additionally, this stage works as a regularization parameter in training. In the end, the model combines the ClassCaps output and the Reconstruction Stage for the optimization weights.

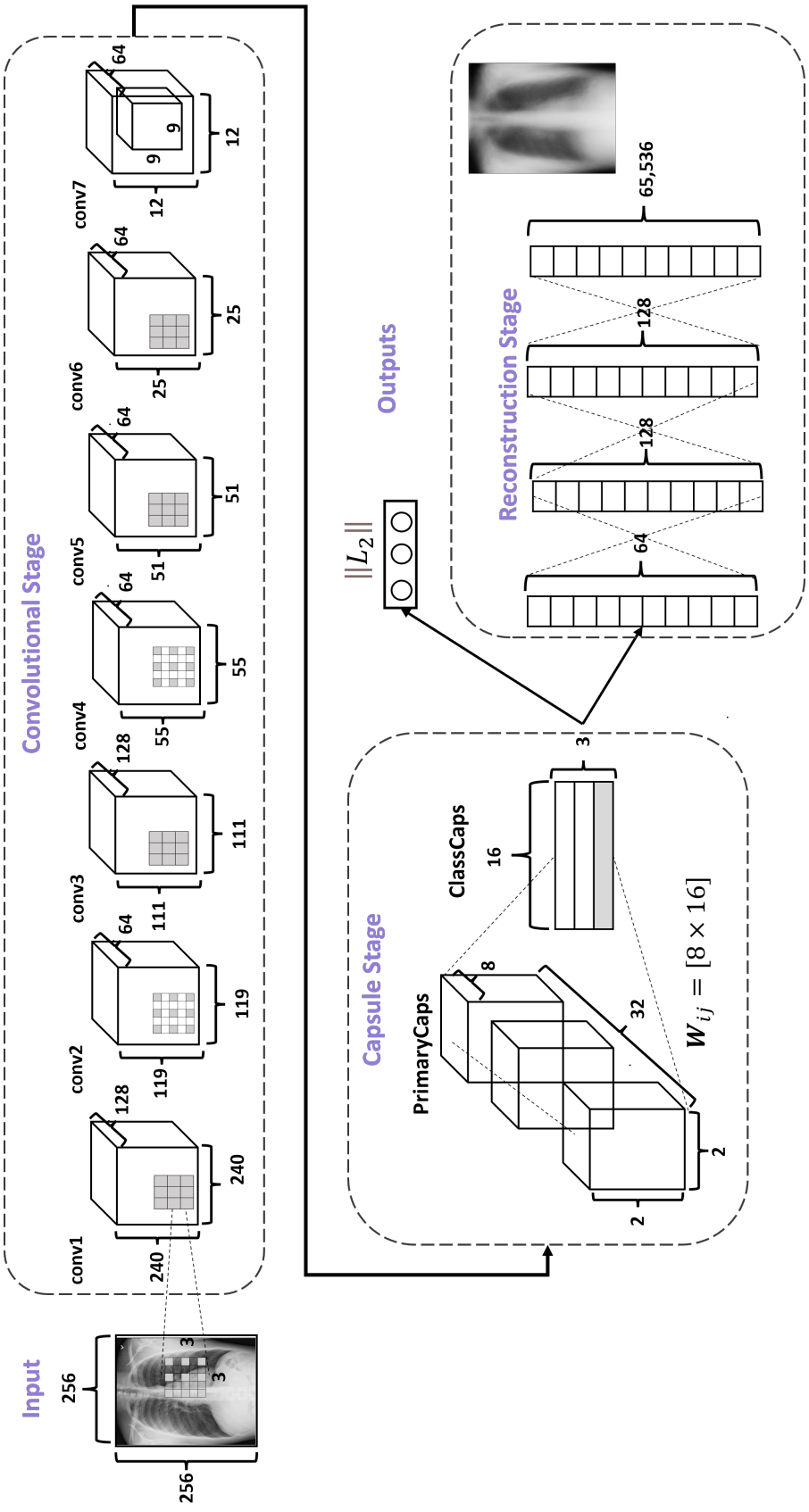


Figure 4.19. DRCaps model.

Convolutional Stage

The Convolutional Stage (CS) converts the intensity of the pixels into activity detectors of local features and uses them as input for the next stage. CS is made up of seven convolutional layers, as shown in Figure 4.20. Layers conv1, conv3, and conv5 use the dilation rate hyperparameter. In the DRCaps model, the conv1 layer applies 128 filters to an input image of 256×256 pixels, using kernels of 3×3 with stride equal to 1 and with a dilation rate equal to $8, 8$. This configuration generates an output size of 240×240 values. As the original goal was to reduce the size of the feature maps, the next convolutional layer (conv2) uses a bigger stride value and only 64 filters. This reduces the size of the feature maps to 119×119 values. So, the next layers change between a convolutional layer with stride equal to 2, and a dilation rate value to reduce the feature maps size and the number of parameters. Finally, the last layer in the CS has 64 filters with a size of 12×12. Additionally, all convolutional layers use the ReLU activation function [69]. Table 4.12 summarizes the selection of hyperparameters on each convolutional layer.

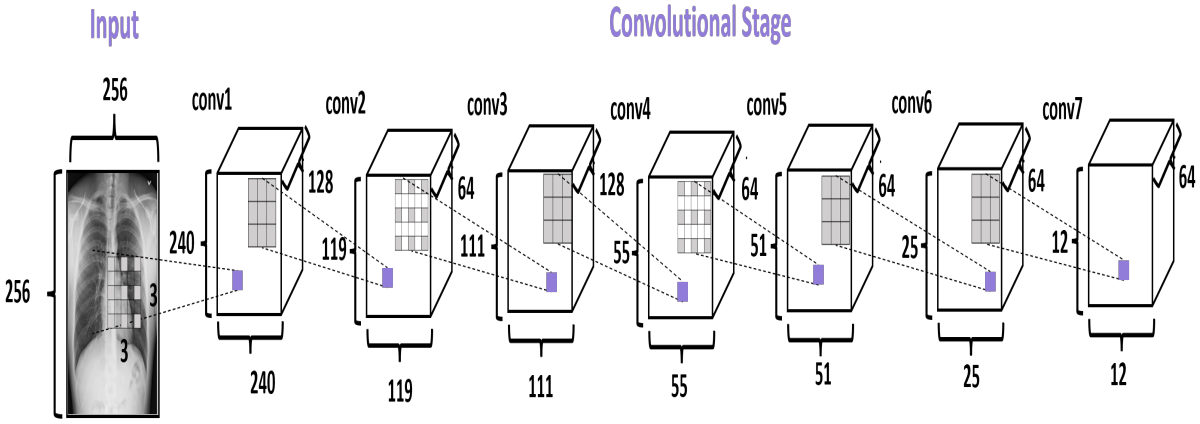


Figure 4.20. Convolutional Stage on the DRCaps model.

Capsule Stage

The Capsule Stage (CaS) consists of two layers: PrimaryCaps Layer and ClassCaps Layer. The PrimaryCaps layer is the lowest level of multidimensional entities from an inverse graph

Table 4.12. Hyperparameter selection on each convolutional layers at the CS.

Layer	filters	kernel value	stride	DR	output value	parameters
conv1	128	3×3	1	(8,8)	240×240	1,280
conv2	64	3×3	2	(1,1)	119×119	73,792
conv3	128	3×3	1	(4,4)	111×111	73,856
conv4	64	3×3	2	(1,1)	55×55	73,792
conv5	64	3×3	1	(2,2)	51×51	36,928
conv6	64	3×3	2	(1,1)	25×25	36,928
conv7	64	3×3	2	(1,1)	12×12	36,928

perspective. This corresponds to a reverse rendering process. Given the 64 filters with the featured map size of 12×12 of the last convolutional layer, a kernel size of 9×9 with a depth of 64 parameters and a stride equal to 2 is applied to the layer, producing 32 PrimaryCaps layers each of size $2 \times 2 \times 8$, where each PrimaryCap layer has 4 capsules of eight dimensions (8D), as shown in Figure 4.21. These operations generate 128 capsules. As explained in Section 4, a capsule is a group of neurons that code the probabilities of feature detection in an output vector. This output vector has two components: magnitude and orientation. The magnitude represents the probability that the entity exists, while the orientation represents the instantiating parameters such as pose (position, size, orientation), deformation, and texture, among others.

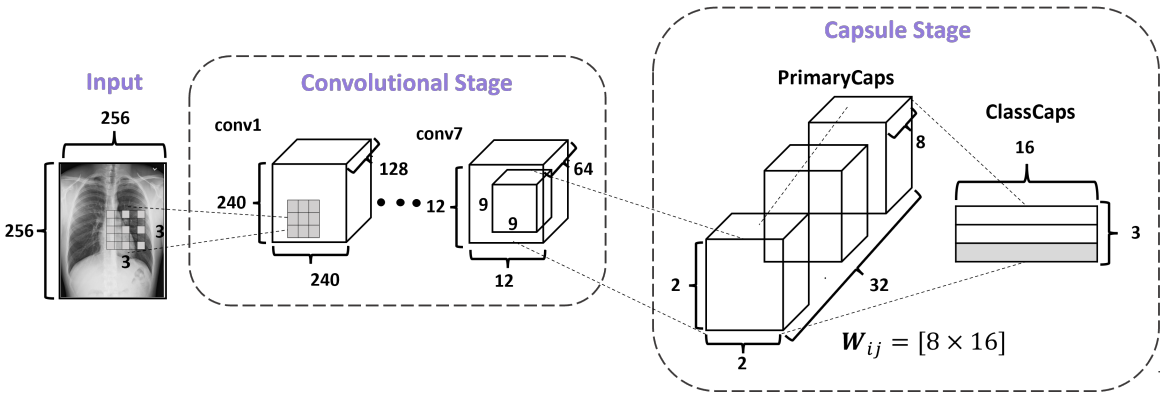


Figure 4.21. The Capsule Stage in the DRCaps model.

Once the capsules are computed, the network decides which information will be passed on to

the next layer. The capsule predictions are made by multiplying each capsule by a weight matrix $\mathbf{W}_{ij}$ for each class we are trying to predict (the COVIDx V7A dataset has three classes). Each weight is actually a matrix with a size of 8×16, so each prediction is a matrix multiplication between the capsule vector and this weight matrix, as can be seen in Figure 4.3. Therefore, we will end up with 384 predictions and each prediction is a 16-degree vector. It is important to note that the 16 dimension is an arbitrary choice, just as 8 is the size of the capsules.

The ClassCaps layer follows the PrimaryCaps layer. The process of changing 384 predictions into three vectors of 16 elements each is performed using the affine transformation matrix (W_{ij}) and the dynamic routing by agreement algorithm explained in Section 4.2. This step is the most important in the architecture because it is the way of how the information is passed to the other layer instead of using max pooling. Furthermore, the ClassCap layer produces two outputs, as shown in Figure 4.19. The first output consists of three vectors produced by the ClassCap layer, where each vector corresponds to each class in the network. Then, this output uses the norm L_2 to calculate the length of each vector. Finally, the vector values are the confidence to detect the associated class, *i.e.*, the prediction. The second output is the reconstruction stage, which we will explain in the next section.

Reconstruction Stage

Finally, the Reconstruction Stage (RS) uses the output of the ClassCap Layer as input to recreate the original input image shown in Figure 4.22. Then the model minimizes the distance (loss) between the reconstructed and original images. Note that, while a traditional CNN cares only about whether the model predicts the correct classification, CapsNets use the reconstruction stage as a regularization method to improve the results with the reconstruction loss. In our architecture, this stage is formed by three fully connected layers of 64, 128, and 128, as shown in Figure 4.22. It is also important to note that, in the reconstruction stage, specifically, the decoder is part of the network that could generate more parameters due to their fully connected layers. In the DRCaps model, this stage generates 8,482,112 parameters.

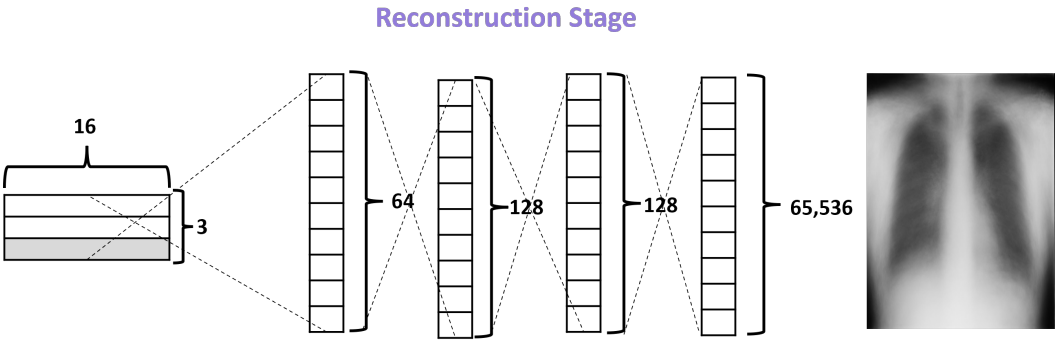


Figure 4.22. Decoder structure of the DRCaps model.

4.8 Chapter Summary

The DRCaps model is based on two architectures: CNNs and CapsNets. This chapter explains in detail the CapsNets architecture and mentions the main CapsNets advantages such as the method of routing information in which the information that is most closely related to the next layer is passed on. This new architecture is highlights in identifying overlapping digits in the MNIST dataset; also offers robustness against adversarial attacks and has the capability that these networks can be trained with less data than CNN-based architectures. However, despite the advantages they offer and the way they work, these networks use a large number of parameters that make the networks impossible to train with complex or large images. This chapter describes how the DRCaps computational model is formed based on the combination of CNNs and CapsNets. The chapter describes the operation of the dilation rate hyperparameter, which is key to the network's ability to handle complex images. Then it describes how this hyperparameter was applied in the original CapNets architecture. Finally, this section describes in detail the DRCaps model.

Chapter 5

Experiments and Results

This chapter begins by introducing the tools used to implement the computational model and to manage the COVIDx V7A dataset, as well as the hardware used for the experiments. Subsequently, the results of the experiments conducted on the DRCaps model are presented. The chapter then discusses the results of the reconstruction stage in the COVIDx V7A dataset before concluding with a discussion of the findings.

5.1 Computational Model Implementation

To create a computational model based on CapsNets that can manage complex images, it is essential to program the original architecture. To do this, the software to be used to program the model must be determined. This step is necessary because, since CapsNets is a new architecture, there are no libraries or high-level Application Programming Interfaces (API) available, such as those for CNN models like AlexNet or VGG.

5.1.1 Machine Learning Software

As the field of machine learning grows, many companies began to create their own libraries to train their models, as illustrated in Figure 5.1. Examples of these libraries include Google's TensorFlow[1], Theano from the Montreal Institute for Learning Algorithms (MILA), and Facebook's PyTorch[2], among others. These libraries are tedious, slow, and inefficient [4].

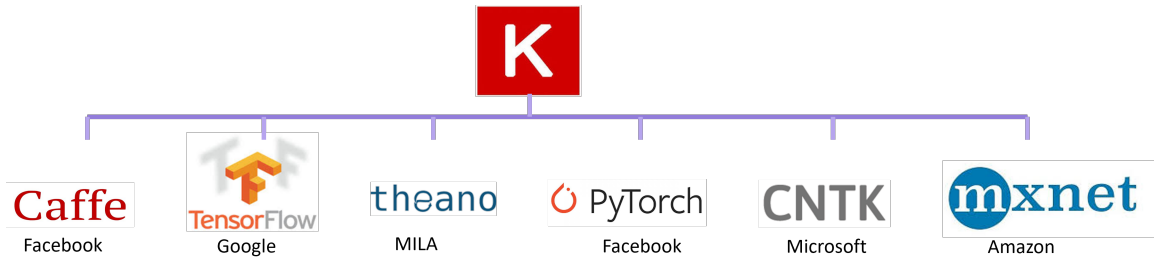


Figure 5.1. Keras backends.

TensorFlow, a library created by Google and released to the public in 2015, is one of the most popular libraries for network training. It is based on Python and is designed to run on CPUs, GPUs, and Tensor Processing Units (TPUs). Additionally, programs written in TensorFlow can be exported to other runtimes, such as C++, JavaScript, or for applications running on mobile devices. However, TensorFlow is complex to program. Then Keras [3] arrives in 2017. Keras is a Python high-level API created by Francois Chollet [4]. Keras stands out for prioritizing the developer experience, which means that it is an API for humans, not machines. Because Keras has a large and diverse user base, it does not force the user to follow a unique way of building and training models. Rather, it enables a wide range of different workflows corresponding to different user profiles.

To work with Keras, it is necessary to select a backend. In simpler terms, a backend is like a database, and Keras is the language that can access that database. Generally, each technology company has its own backend. Therefore, Keras was designed to work with different backends,

[1] https://www.tensorflow.org/
[2] https://pytorch.org/
[3] https://keras.io/

as shown in Figure 5.1. At the beginning, Theano was the official backend for Keras until version 1.1.0. After that, TensorFlow (TF) became the default backend, but both frameworks had to be installed separately, as their versions were constantly changing. In 2019, the latest version of Keras (2.3.0) was released, which supports multiple backends. All subsequent versions work only with TF. At the same time, TF changed to its 2.0 version, which designated Keras as its official high-level API. This caused several changes, such as: the two versions being installed together, some functions being absorbed by other functions or marked as deprecated, and the libraries needing to be called differently. It is important to be aware of these updates as it is necessary to know which versions of Keras or TensorFlow the programs were created for in order to simulate them or make the necessary updates in the programs so that they run in the desired version.

Because the Keras API is built on top of TensorFlow, any type of Deep Learning model can be defined and trained. For example, the DRCapsNet model is formed in different stages. Each stage must be programmed differently depending on the needs of the architecture. The reconstruction stage is the easiest part of the model to program because it uses already defined layers, such as Fully Connected ones. On the other hand, the Capsule Stage requires one to create new libraries to implement the PrimaryCaps layers and the new loss function (margin loss), because there are not exist in the actual Keras and TensorFlow libraries.

Therefore, to build the DRCapsNet model, it was necessary to analyze the three options offered by Keras to build a model: Sequencial Model, Functional API, and Model Subclassing. The three options are explained next.

- **Sequencial Model**: This option is appropriate for a plain stack of layers where each layer has exactly one input and one output. But a sequential model is not suitable when: the model has multiple inputs or multiple outputs, or any of the layers has multiple inputs or multiple outputs, or it is necessary to do layer sharing. This option was used in the decoder stage to reconstruct the predicted image.
- **Functional API**: This option is a way to create models that are more flexible than the Sequential Model, this API can handle models with shares layers or multiple inputs or outputs. This means that you can create a model by specifying its inputs and outputs in the

layer graph. Also, with the Functional API when the model has two outputs, it is possible to assign different losses to each output, and it is possible to assign different weights to each loss to modulate their contribution to the total training loss. This is the configuration used to build the CapsNet model because we have two output models: a training model and an evaluation model, as shown in Figure 4.11. In our model, we use the margin loss and the *mae* loss (see Equation 4.4). Also, something useful in this configuration is that as the outputs have different names, it is possible to specify the losses and loss weights with the corresponding layer names. In addition, this configuration was used because it includes a wide range of built-in layers, for example, convolutional layers, pooling layers, and useful tools such as batch normalization, dropout, etc.

- **Model subclassing**: This option helps to build new layers, where it was necessary to implement two methods: *i)* the call method that specifies the computation done by the layer and *ii)* the build method, which created the weights of the layer; also it is possible to create the weights in the initial method as well. In our model, it was necessary to create our own three layers: *i)* the capsule layer is similar to the dense layer, but in this layer the dynamic routing was performed by agreement procedure. ii) the length layer; here it computes the length of the output vectors to compare the predicted output with y_{true} for the margin loss. iii) also the mask layer was created; here the layer masks a tensor either by the capsule with maximum length or by an additional input mask in the case of training. Except for the max-length capsule, all vectors are masked to zeros. Then flatten the masked tensor for the decoder model.

Keras does not limit users to one type of model; instead the functional API, the Model subclassing, and the Sequential models can all be used together. This thesis will use all three of these options to create the DRCaps computational model. The programming codes used in this work are presented in Appendix A.

5.1.2 Computational Model Flow Chart

The flow chart in Figure 5.2 illustrates the process of the experiments carried out in each version of the computational model. It begins by loading the input arguments and the training dataset into memory according to the batch size specified in the input arguments. Then, the model is constructed, specifying the number of layers and the hyperparameters associated with it. After that, the compilation parameters are set. Once the model is configured, training can begin. After the training is complete, the model shows the training loss graphics and the weights are saved. Subsequently, the testing dataset is loaded and the model is tested to obtain the final accuracy and the reconstruction graphics. The code for this process is provided in Appendix A.

5.2 Experimental Platform

5.2.1 Hardware

Most of the experiments were carried out on a server at Centro de Investigacion en Matematicas [4] (CIMAT), called Tinieblas. The connection was made through public JupyterHub, where we signed in, and then we worked in a Jupyter Lab session. The Tinieblas server has three GPU cards: two GeForce RTX 2080 Ti with 11 GB VRAM and a GeForce RTX 2080 with 8 GB VRAM, which has a Compute capability (CC) equal to 7.5. The CC identifies the features supported by the GPU hardware and is used to determine which hardware features and/or instructions are available on the present GPU [5]. The Deep Learning framework used was Keras 2.5.0 with TensorFlow 2.5.0 as a backend, as shown in Table 5.1.

Table 5.1. Experimental platform.

Server	OS	TensorFlow	Keras	GPU	CC
Tinieblas	Linux	2.5.0	2.5.0	RTX 2080 Ti	7.5

[4] https://www.cimat.mx/

[5] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

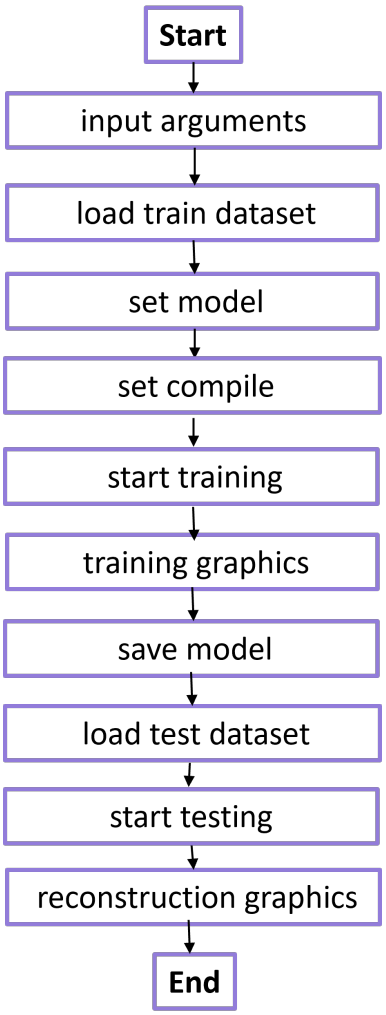


Figure 5.2. CapsNet Model flow chart.

5.2.2 Dataset Adjustments

The COVIDx V7A dataset, which contains real images with high resolution, poses a challenge for a CapsNet model during the training stage due to its high memory requirements. In comparison, the MNIST dataset [38] requires minimal memory usage, since it consists of 60,000 training images, each with a dimension of 28 $\times$ 28 pixels, which is a total of 47 MB (Megabytes). However, the COVIDx V7A dataset, despite having only 15,000 training images, has a resolution of 1024 $\times$ 1024 pixels, which requires a space of a little more than 1 MB per image, making it very difficult to load the entire dataset into memory. Therefore, we decided to use an image size of

256 × 256 using the bilinear method for the input images. Additionally, to address this issue, we initially attempted to divide the training process into smaller blocks. To do this, we used the Keras ImageDataGenerator class, which allows us to create these small blocks and, at the same time, perform data augmentation in real-time. However, as Keras is on top of Tensorflow, programs with the Keras API are more straightforward than other APIs but, at the same time, are more limited when customizing an algorithm and can be slower. For these reasons, TensorFlow 2.0 offers an API that can help with this task, the tf.data API [171]. This API allows us to handle large amounts of data, read from different data formats, and perform complex transformations in a fast and scalable way. The main component that it uses is tf.data.Dataset which can be used for the following tasks: *i)* Creating a dataset object from input data, *ii)* Applying dataset transformations for preprocessing, *iii)* Iterating the dataset in a streaming fashion and processing the elements. Thus, the reason for choosing tf.data over ImageDataGenerator is that generating training and validation batches with tf.data is much faster than ImageDataGenerator. Several experiments report that tf.data is about 34 times faster than ImageDataGenerator [171].

To use the tf.data API in our model, the COVIDx V7A dataset must be restructured to have three subfolders, each with the name of a class and the corresponding images, as shown in Figure 5.3. The original COVIDx V7A dataset had only two image folders, labeled 'train' and 'test', where all the images were without being separated by their class. Furthermore, the dataset contained two text files, where each line provided the following information per image: patient ID, filename, class, and data source for each image.

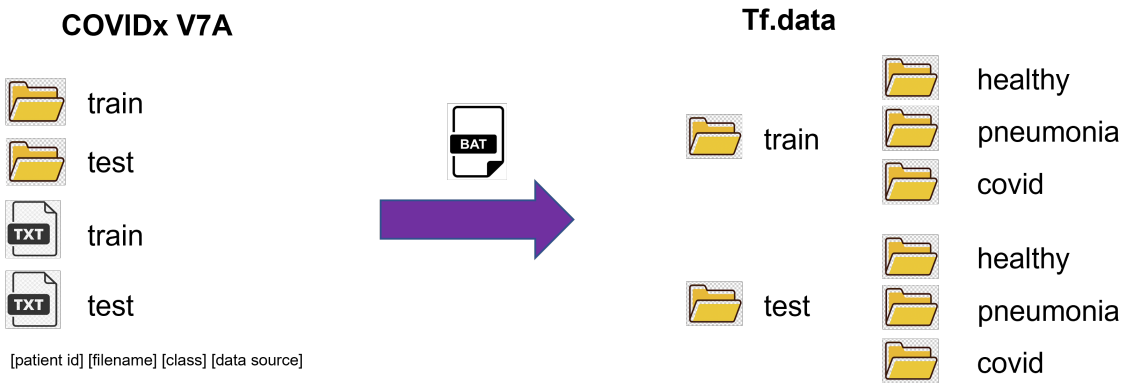


Figure 5.3. COVIDx V7A dataset organization for tf.data API.

5.3 Results of the DRCapsNet Model Versions in MNIST Dataset

Table 5.2 lists the hyperparameters used in the experiments to train the DRCaps models with the MNIST dataset. It should be noted that a hyperparameter is a parameter of the training algorithm (not of the network), meaning that its value is not altered by the training. It must be established prior to training and kept constant during the training process.

Table 5.2. Hyperparameters used for the MNIST dataset .

Arguments	**Compilation**	**Training**
epochs = 50	optimizer = Adam	train images = 60,000
learning rate=0.001	learning rate= 0.001	validation images= 10,000
img width=28	prediction loss= margin loss	batch size=100
img height=28	reconstruction loss= mse	test images= 10,000
lr decay=0.9	metrics= accuracy	
routings=3		
λ recon= 0.392		

Table 5.3 presents a summary of the experimental results of the different DRCapsNet models depicted in Figure 4.17. It can be seen that DRCapsNet V2 overcomes in accuracy the other models on the MNIST dataset. Despite the fact that the accuracy seems to be almost the same in all three versions. This soft variation has an important impact on the reconstruction stage, as can be seen in Figure 5.4.

The first column of Figure 5.4 shows the original CapsNet model which was obtained in the first training of the model, which yielded an accuracy of 89% due to the random initial parameters. When the same version was retrained, it gave an accuracy of 98%, which is reported in Table 4.11. We include this version because it is very interesting to see how the network struggles to reconstruct the original input even when the accuracy is not a bad result. Also, it can be seen that,

Table 5.3. Accuracy (acc) results at different dilation rate values.

Network	DR	acc (%)
CapsNet	no	98.5
DRCapsNet V1	(2,2)	99.2
DRCapsNet V2	(1,1),(2,2),(4,4)	**99.6**

as the accuracy improves, the reconstructed digits do the same. This can be seen in the digits in orientation or sharpness. Finally, Figure 5.5 shows the training results for each version. We trained up to 50 epochs because those were the epochs used reported in other articles [89, 172]. However, it is observed that only 20 or 15 epochs are enough to obtain a final result. This was taken into consideration for the following experiments.

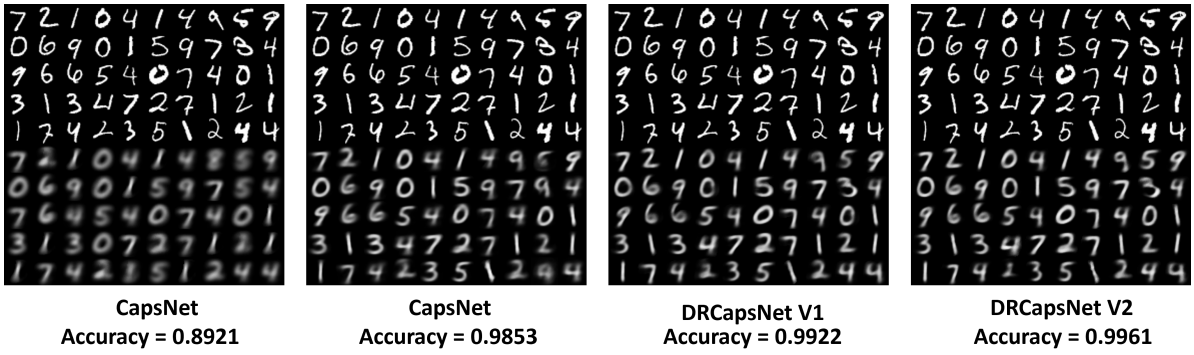


Figure 5.4. The first five lines of each figures are the input images to the architecture, and the last five lines show how the network reconstructed them.

Now, Figure 5.6 shows the confusion matrix of the DRCapsNet V2 model. Like MNIST is a benchmark dataset, all its classes are balanced. It can be observed in the matrix that the digit that has more missclassifieds is number 9, and the easiest to classify is number 0. Perhaps to improve missclassification, a solution could be to create more number 3 digits, but instead of using data, the instantiation parameters learned by the network could be increased, as shown in Figure 4.8. The results of these experiments suggest that the dilation rate hyperparameter can be used in capsule networks in ConvLayer to reduce the number of parameters and capsules, potentially leading to an improvement in the accuracy of the network.

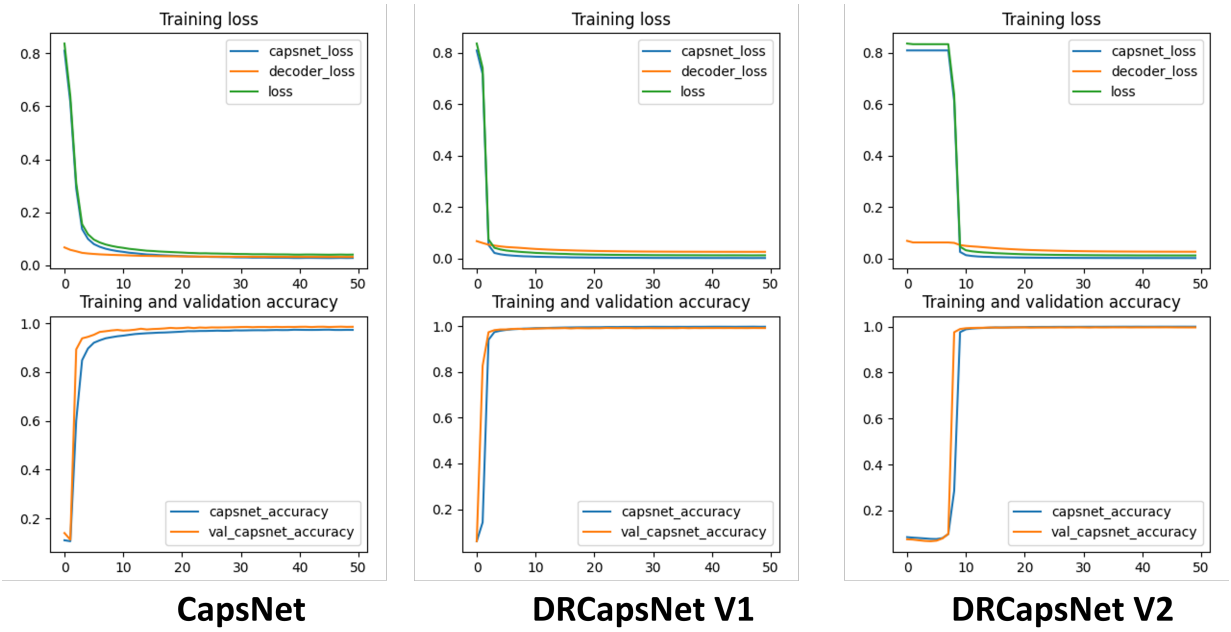


Figure 5.5. MNIST training results.

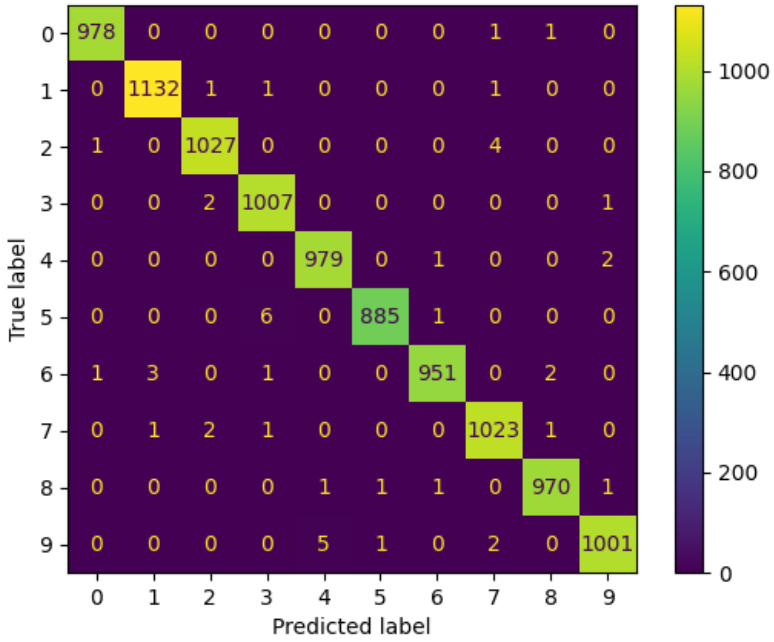


Figure 5.6. MNIST dataset confusion matrix.

Table 5.4 shows the metrics calculated for each class of the MNIST dataset. As you can see, the metrics are almost ideal because this dataset is balanced and the input images are very easy to classify.

Table 5.4. DRCapsNet V2 metrics results on MNIST dataset.

	precision	***recall***	***F1-score***	**images**
0	1.00	1.00	1.00	980
1	1.00	1.00	1.00	1135
2	1.00	1.00	1.00	1032
3	0.99	1.00	0.99	1010
4	0.99	1.00	1.00	982
5	1.00	0.99	0.99	892
6	1.00	0.99	0.99	958
7	0.99	1.00	0.99	1028
8	1.00	1.00	1.00	974
9	1.00	0.99	0.99	1009
accuracy	99.60 %			

5.4 Results of the DRCapsNet Model Versions in COVIDx V7A Dataset

Table 5.5 lists the hyperparameters used in the experiments to train the DRCapsNet models with the COVIDx V7A dataset. Next, Figure 5.7 shows the training loss functions of the DRCaps model up to 50 epochs. However, we can observe that only 20 are enough to obtain a good performance. The red line is the reconstruction loss (mae), the green line allows the loss of CapsNet (margin loss), and the blue line shows the total loss function, which is formed by adding the loss of CapsNet and the loss of the decoder increased by the reconstruction value of λ.

Figure 5.9 shows some examples of the output of the reconstruction stage in DRCapsNet.

Table 5.5. Hyperparameters used for the COVIDx V7A dataset.

Arguments	**Compilation**	**Training**
epochs = 50	optimizer = Adam	train images = 12,089
learning rate=0.001	learning rate= 0.001	validation images= 3022
img width=256	prediction loss= margin loss	batch size=32
img height=256	reconstruction loss= mae	steps per epoch= 378
lr decay=0.9	metrics= accuracy	test images= 1,579
routings=3		
λ recon= 40		

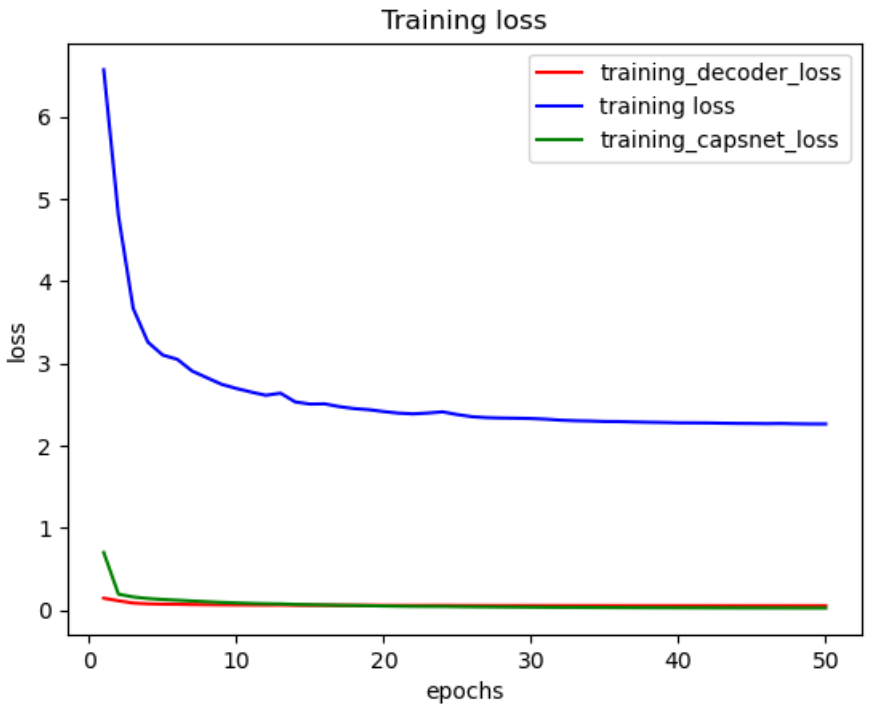


Figure 5.7. Reconstruction loss of the DRCapsModel.

These results correspond to $\lambda = 40$, which produced an accuracy of 90%. The first four rows of Figure 5.9 are random examples of the images entered into the model, and the last four rows show how the network attempts to reconstruct them. As can be seen, the network focuses on the chest section and attempts to recover the size, position, and shape of the lungs. On the other hand, Figure 5.8 shows the output of DRCapsNet model. These results correspond to $\lambda = 0.392$, which produced

an accuracy of 10%. In this case, the regularization parameter λ is small and does not enforce a correct reconstruction of the input images.

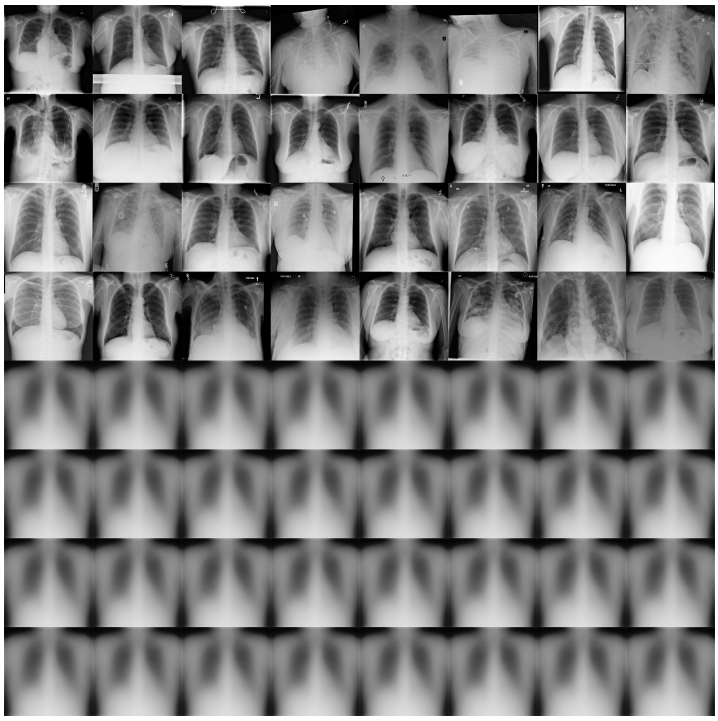


Figure 5.8. Reconstructed COVIDx V7A images from RS in the DRCaps model with $\lambda = 0.392$.

Figure 5.10 shows the confusion matrix of the DRCapsNet model in the COVIDx V7A dataset. The DRCapsNet model achieves an accuracy of 90%. The confusion matrix shows us that the class that the model classifies most accurately is healthy and the class that classifies worst is covid. This makes sense because in both the training and test data, the healthy class has a greater number of images, and the covid class has the smallest number. Unlike the MNIST dataset the COVIDx V7A have unbalanced classes in their training and testing data, as shown in Table 2.2. For this reason, it is necessary to consider another metric in addition to accuracy.

Table 5.6 shows the metrics results for each class from the confusion matrix shown in Figure 5.10. As can be seen in Table 5.6, the covid class is the one that offers the worst results. The *precision* metric tells us of all diseases classified as covid, which truly belonged to this class. On the other hand, the *recall* metric tells us of all the images that really belong to the covid class, which ones were correctly classified. Finally, the F1 score metric combines the values of the two previous metrics, giving them the same importance, and is widely used when we have an unbalanced dataset

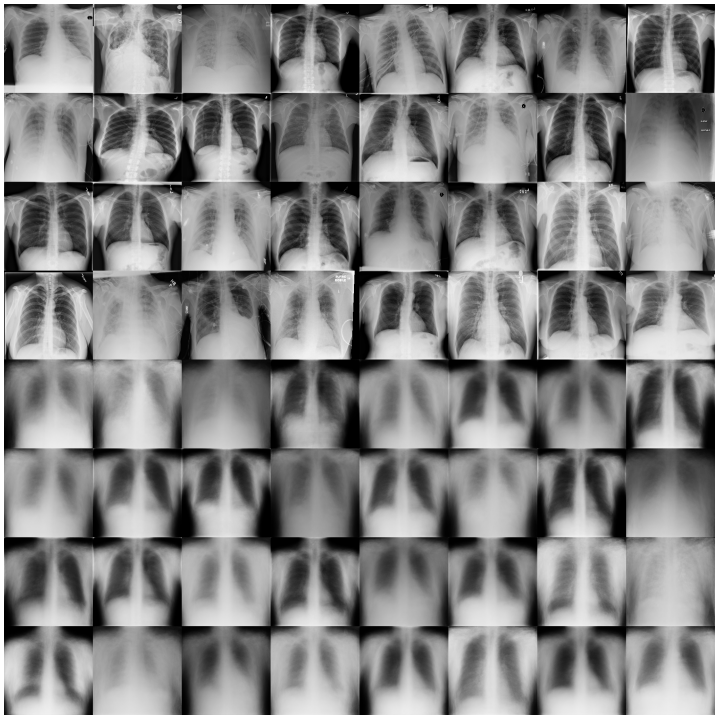


Figure 5.9. Reconstructed COVIDx V7A images from RS in the DRCaps model with $\lambda = 40$.

as is our case. Here, it is observed that the healthy and pneumonia classes have good performance while the covid class does not, despite obtaining good accuracy in the model. One of the reasons for this imbalance of results is due to the small number of images available from the covid class both for training and testing the model. For this reason, it was decided to conduct an experiment with a balanced dataset to see the behavior of the model. The balanced dataset used is a reduction of the COVIDx V7A dataset. What was done was to leave the same number of training and test images for each class. As mentioned previously, the covid class is the one with the fewest images, so the number of images per class of the new dataset was adjusted to the same as shown in Table 5.7.

Table 5.7. COVIDx V7A balanced dataset classes.

Data	**Pneumonia**	**Healthy**	**COVID**	**Total**
train balanced	1670	1670	1670	5,010
test balanced	100	100	100	300

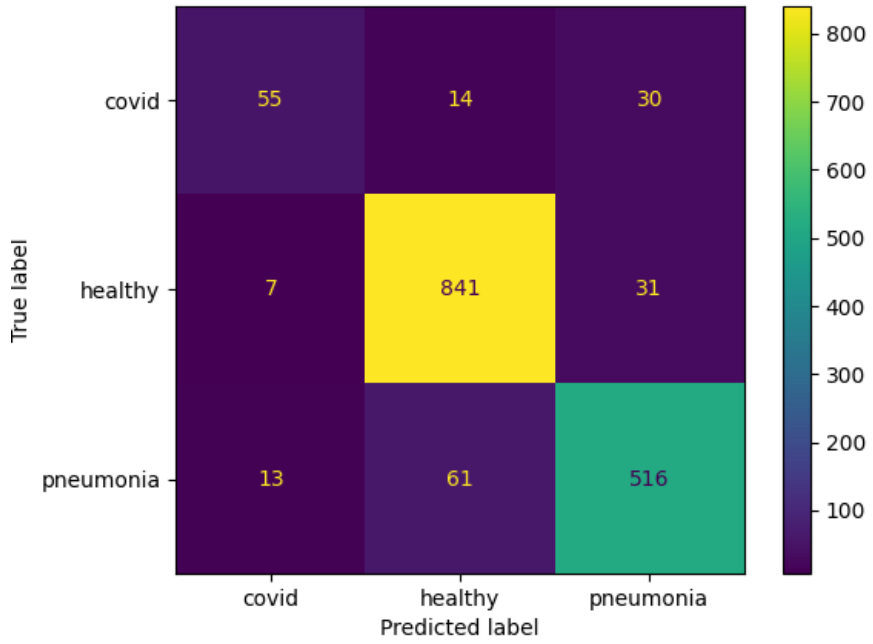


Figure 5.10. DRCapsNet model confusion matrix in the COVIDx V7A dataset.

Table 5.6. DRCapsNet V metrics results.

	precision	***recall***	***F1-score***	**images**
covid	0.73	0.55	0.62	99
healthy	0.91	0.95	0.93	879
pneumonia	0.89	0.87	0.88	590
accuracy	90.05 %			

The results of this experiment are shown in Figure 5.11 and Table 5.8. The accuracy of the model is 72. 56 %. It can be seen from the confusion matrix (Figure 5.11) that the model continues to struggle to correctly classify the covid class with the pneumonia class. However, if you look at the results of the F1 score metric, you can see that the difference in the results by class is no longer as significant as in Table 5.6. It should be noted that the DRCapsNet model in the COVIDx V7A dataset, the F1 score value of 0.93 for the healthy class and 0.88 for the pneumonia class, was achieved with 7,966 and 5,474 training images, respectively. However, a value of 0.62

was obtained with 1,670 training images. Now if we compare the results of the model F1 score metric in the COVIDx balanced dataset, the values of the healthy and penumonia classes decrease considerably while the value of the covid class almost remains the same. This shows us that the number of 1,670 images per class for training the model is not enough to achieve acceptable performance for the DRCapsNet model. However, if we increased the number of images of the covid class to the number of images of the pneumonia or healthy class of the COVIDx dataset unbalanced, the model would considerably increase the value of the F1-score of the covid class and, therefore, the accuracy of the model. Unfortunately, new data are not always available to add to the dataset, especially in recently created datasets such as our case.

Table 5.8. DRCapsNet V metrics results in the COVIDx V7A balanced dataset.

	precision	***recall***	***F1-score***	**images**
covid	0.82	0.55	0.66	98
healthy	0.75	0.82	0.78	93
pneumonia	0.66	0.81	0.73	97
accuracy	72.56 %			

DRCaps Discussion

One of the main problems with CapsNets is that they struggle when handling complex images because the network wants to understand everything about the input image. Thus, the region of interest is a small fraction of the input image in order to produce a correct classification. This results in very time-consuming training and a decrease in accuracy.

For these reasons, CNNs continue to be used as the first stage of the model for feature extraction, as they have demonstrated their efficient performance in image classification. The selection of parameters in each convolutional layer is essential because the information obtained is the one that will form the capsules in the next stage. For example, if we do not reduce the size of the feature maps resulting from the CS, when entering the CaS, the number of capsules will be very large. Some articles that work with large images use the max-pooling operation to do this

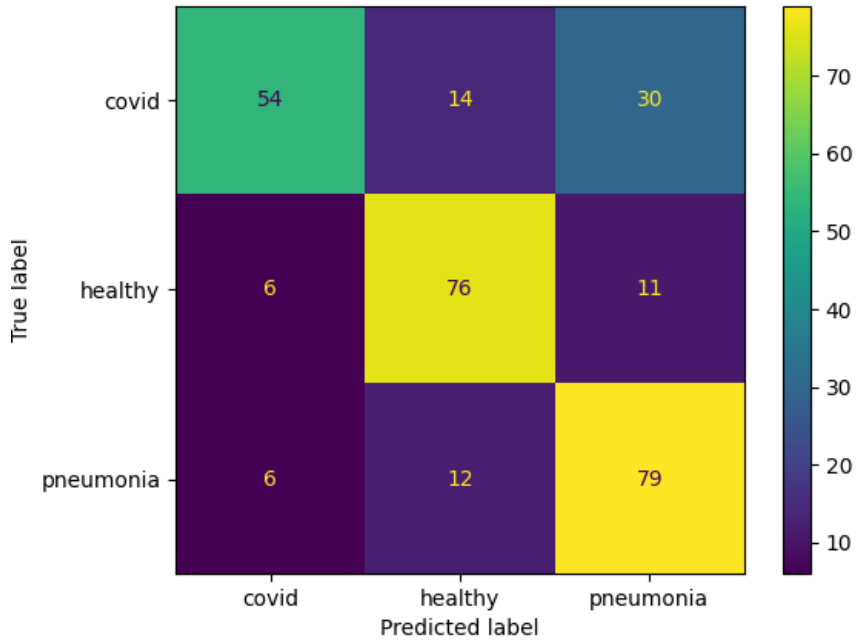


Figure 5.11. DRCapsNet model confusion matrix in the COVIDx V7A balanced dataset.

size reduction, even though it causes a loss of information. In our model, instead of using this operation, we decided to use dilation convolution (controlled by the dilation rate hyperparameter) to cover a broad field of the image at a lower computational cost, in addition to using the stride parameter to reduce the size of the feature maps.

At the beginning of our experiments, we only used the dilation rate parameter, which generated large feature maps in the last convolutional layer; this caused the network to have many capsules, which caused slow training and low accuracy. The reason may be that the capsules tried to codify and reconstruct all the details of the image, but much of that information is irrelevant to our task. According to the experiments, there is a relationship between the number of capsules and the complexity of the images to analyze. For example, for the original CapsNet architecture, which uses an image size of 28×28, its model can handle 1,152 capsules with reasonable accuracy. However, when we tried to use an excessive number of capsules, such as 2,000 capsules, 18,432 capsules, and 67,712 capsules, the accuracy decreased to 60%, 71%, and 10%, respectively. For these reasons, we reduced the number of capsules by manipulating the convolutional kernel stride

to improve accuracy. Figure 4.20 and Table 4.12 show how the dilation rate and stride reduce the size of the filters in the CS. This results in creating a smaller number of capsules and a smaller number of total parameters in the network.

In CaS, we maintain the same arbitrary parameters as in the original paper, such as the dimension of the capsules equal to eight, the size of the ClassCap layer as a 16 D capsule per class, and three dynamic routing iterations. We tried to adjust this parameter manually, but had no significant results.

Finally, we observed that RS requires a larger number of parameters (weights) in the network, and some implementations (reported works) prefer to omit this stage. However, this stage is essential to train the network because it enforced the model to improve their feature extractor to codify all the information in the original image.

Our observation is consistent with reported works that modified RS to improve accuracy [89, 86, 7, 87, 102]. We note that the weight λ of the reconstruction loss significantly affects the result of our model and prevents overfitting. Furthermore, the experiments show a correlation between the input image size and the value λ: we set it equal to the proposed value of 0.0005 (original paper) multiplied by the size of the images used. Once you have this reference value, you can play with the values up or down, observing the behavior of the network. From Table 4.11, we can see that in our case, the reference value with which we started was $\lambda = 32,768$. Then, we tried to increase the value to $\lambda = 40$ which resulted in an increase in accuracy, so we decided to double the value to $\lambda = 80$ and produced a decrease in the accuracy obtained. We trained our model with a modest number of examples because CapsNets are capable of generalizing using much less data in contrast to other CNNs that require a large amount of reference data for the training phase [89].

Table 5.9 shows the advantages obtained with the DRCapsNet model and compares the model with similar input data architectures. The data used in the models in Table 5.9 are from X-rays, Compute Tomography (CT), Magnetic Resonance Imaging (MRI), and Automated Breast Ultrasound (ABU). Also, the architectures in Table 5.9 are ordered with greater accuracy. Although our model did not reach the first place in Table 5.9, we can say that it offers a more robust architecture. For example, the first place in the table uses images smaller than ours and only

has two classes, simplifying the task's complexity. The second place at the table handles similar image sizes, but they use only two classes and do not include a reconstruction stage because they use color images. Also, they still use the max pooling operation on the CS. Although the Kruthika et al. model achieves an accuracy of 94.06% using three classes, their input images have smaller sizes and can have a larger number of capsules than ours. That is a limitation for implementing a detector of other diseases where the features would present high spatial frequency characteristics or more classes need to be detected. Also, some articles say that with their computing resources, it is impossible to use CaspsNets for high-quality images. For the same reason, other papers avoid the reconstruction stage to design a lighter model. Furthermore, few publications use CapsNet with image sizes greater than 128×128 pixels.

5.5 Chapter Summary

In this final chapter, we describe the implementation of the DRCapsNet computational model. It explains the reasons for the selection of software and hardware. Also, explains the modification of the dataset needed for the COVIDx V7A dataset in order for the network to use it for training. Then, it presents the results of the validation of the CapsNet original architecture in the MNIST dataset with the dilation rate hyperparameter. The results from this experiment outperform the state-of-the-art results, see Table 4.5.

Finally, it presents the results of the validation of DRCaps on a complex medical dataset (COVIDx V7A). The results show interesting results associated with some model hyperparameters. For example, it highlights the relationship between the number of capsules and the complexity of the images to analyze and the correlation between the size of the input image and the hyperparameter λ. The experimental results show that our model obtains an accuracy of 90%, which is an acceptable performance compared to other deep learning architectures. The contributions and future research opportunities for the DRCaps model are explained in detail in the next chapter.

Table 5.9. Accuracy results of different CapsNets-based models in medical imaging. Some models do not have defined some parameter and are listed as not mentioned (nm).

Study	Model	Data	Input	Channels	Capsules	Classes	Decoder	Recons.	Max-Pooling	Data Aug.	Accuracy (%)
Ali et al. [173]	19-layers CNN	X-ray	227 × 227	3	nm	2	nm	no	yes	no	98.5
Mittal et al. [165]	ECC	X-ray	100 × 100	1	36,864	2	2	yes	no	no	95.9
Afshar et al. [167]	COVID-CAPS	X-ray	224 × 224	3	nm	2	nm	no	yes	no	95.7
Kruthika et al. [28]	CapsNets	MRI	64 × 64	1	18,432	3	3	yes	no	no	94.0
this	DRCapsNet	**CT**	**226 × 226**	**1**	**128**	**3**	**3**	**yes**	**no**	**no**	**90.0**
Mobiny [25]	Fast CapsNet	CT	32 × 32	1	2048	2	nm	yes	no	no	88.5
Khanna et al. [166]	DECAPS	CT	448 × 448	nm	nm	2	nm	no	no	yes	87.6
Sarki et al. [174]	VGG-16 CNN	X-ray	224 × 224	3	nm	3	nm	no	yes	no	87.5
Afshar et al. [26]	CapsNets	MRI	64 × 64	1	18,432	3	3	yes	no	no	86.5
Xiang et al. [164]	3-D ResCapsNet	ABUS	128 × 128	nm	nm	2	nm	no	no	nm	84.9
Toraman et al. [168]	CapsNets	CT	128 × 128	1	8192	3	4	yes	yes	yes	84.2

Chapter 6

Conclusions and Future Work

In this thesis, we have described the process of building a computational model based on Capsule Networks for the image classification task, specifically for a medical dataset with high resolution images. The proposed model was built by combining the advantages of Convolutional Neural Networks (feature extraction) with the advantages of Capsule Networks (routing information). As a result of this network combination, different aspects of the computational model were formally analyzed, leading to the following contributions:

- The proposal to grouping into four categories the CNN limitations reported across the literature in many real-world applications.
- The combination of the dilation rate and stride hyperparameters to replace the max-pooling operation allowing the network to handle more complex images.
- The relationship between the number of capsules and the complexity of the images to analyze.
- The correlation between the size of the input image and the value λ.
- Implementing a computational model based on CapsNets capable of being trained with a complex dataset.

In this chapter, we discuss the main findings obtained as a result of this thesis and provide directions for future research opportunities.

6.1 Main Findings

This thesis has presented the development of a novel computational model based on CapsNets. This model is able to handle datasets with more complex images than those previously used in other CapsNet-based computational models.

To construct the model, it was essential to have a deep understanding of the CapsNet approach to address all the issues associated with the implementation of the computational algorithm outlined in Section 5.1.1. Additionally, it was necessary to overcome the issue of memory saturation when dealing with high-resolution images, both in the loading of the images to the algorithm and in the flow of the images in the computational model. Furthermore, different experiments were conducted to enhance the performance of the network. This involved an analysis of some modifiable hyperparameters of the model (DR, Conv Stage, depth, kernel size, featured maps) and training (loss function, λ). This section will discuss the main discoveries made during the model building process.

The experiments have demonstrated that a small number of capsules must be managed for the model to be trainable. To do this, the size of the featured maps in the last layer of the CS must be regulated, as the number of capsules to be formed in the next stage is determined by these values. To control the size of these layers, different sizes of DR and stride hyperparameters can be employed.

It is essential to analyze the decoder architecture in the reconstruction stage when constructing a computational model based on CapsNets. This stage produces a large number of parameters that can overwhelm the memory. The architecture should be configured with the least amount of parameters needed to reconstruct an image with the same quality as the input image in a satisfactory manner.

Another main finding is that selecting the value of the λ hyperparameter is a key feature of our model. This single value can improve the accuracy of the network and prevent overfitting. Experiments have shown that the best approach is to begin with the product of 0.0005 and the size of the input image. After that, it is possible to adjust the value slightly, an excessive increase can lead to a decrease in accuracy, as demonstrated in Table 4.11.

Also, it should be noted that the DRCaps model is a robust architecture, because it requires minimal preprocessing of the input images, does not use data augmentation or weight vectors, and still achieves satisfactory accuracy compared to other architectures, as demonstrated in Table 5.9.

6.2 Main Challenges

Creating a computational model from scratch is a complex task, as it requires taking into account a variety of variables at both the software and the hardware levels. One of the most important things is having a NVIDIA GPU for the training of the model, as it can significantly reduce the training time. To ensure the correct installation of the drivers, it is necessary to first check the Compute Capacity (CC) of the graphics card. From these values, the library (CUDA SDK) supported by the card must be selected and matched with the TF version used. After that, the GPU drivers, CUDA toolkit and CUDNN library must be installed.

An additional challenge was to carefully evaluate the needs of the computational model in order to select the most appropriate library for its implementation. For example, DRCaps had three distinct requirements: i) it had one input but generated two outputs, ii) it is necessary to develop the CapsLayer, DigitCaps, and the dynamic routing by agreement algorithm from scratch, and iii) the decoder part had to be included. As a result, the Keras framework was chosen because it allows the implementation of a computational model in multiple ways, all of which can be incorporated into the same program.

Another challenge presented in this thesis was handling a real data set. Unlike the simple datasets used as benchmarks, such as MNIST or CIFAR-10, which are usually preloaded into machine learning libraries, a new dataset requires analysis of how to call it to the algorithm,

optimizing the computational resources, and adapting it to the requirements of the library used, as explained in Section 5.1. Furthermore, it is necessary to analyze how information can be divided or preprocessed according to the needs of the algorithm.

6.3 Future Work

Once the CapsNet-based computational model is implemented and a tool for dealing with complex images is established, numerous research possibilities can be identified to expand upon this project. Subsequently, some of these research opportunities will be summarized.

In this thesis, we tested the effectiveness of the dilation rate hyperparameter as a tool for a CapsNet to process complex images, which is one of the main limitations of these networks. We used the DRCaps model with input images of 256 x 256 pixels, although the original size of the dataset was 1024 x 1024 pixels. We could conduct an experiment to gradually increase the image size and adjust the dilation rate parameter to determine how far the network can handle high-resolution images.

The CapsNet architecture has the benefit of being able to generalize results, meaning that it can achieve the same outcome as a CNN using only a fraction of the data. To further explore this, an experiment could be conducted to determine the minimum number of input images needed to obtain an acceptable accuracy when using both a convolutional network and a capsule network.

Other research opportunitie is that the COVIDx V7A dataset has a major downside in that its classes are unbalanced. This is a common issue in real and novel datasets. A potential solution to this problem could be to use the instantiation parameters obtained through training in the ClassCap layer to balance the dataset, instead of relying on traditional methods such as data augmentation or weighting.

In the medical field, the overlapping digit recognition capabilities discussed in Section 4.4 can be used to create a system that not only looks for a single disease, but also identifies potential overlap diseases that other models may overlook. This improvement could help the radiologist

make a more comprehensive diagnosis in a shorter period of time. In addition, the algorithm would suggest other diagnoses that would not normally be the cause of a consultation.

It is evident that CapsNets are a more advantageous approach than the current architectures, yet further research is needed before they can be implemented in advanced fields. The fact that basic capsules can provide satisfactory results with minimal preprocessing is a sign that the CapsNet architecture is worth exploring in greater detail.

Bibliography

[1] L. Fei Fei, Y. Serena, and J. Justin, *Standford CS class CS23: Convolutional Neural Networks for Visual Recognition*. http://cs231n.stanford.edu, 2017.

[2] A. Alexander, *MIT Introduction to Deep Learning 6.S191*. http://introtodeeplearning.com, 2023.

[3] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[4] F. Chollet, *Deep Learning with Python*. Manning, Nov. 2021.

[5] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[6] N. Bourdakos, "Understanding capsule networks — ai's alluring new architecture," February 2018.

[7] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in Neural Information Processing Systems*, 2017, pp. 3856–3866.

[8] G. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with em routing," 2018. [Online]. Available: https://openreview.net/pdf?id=HJWLfGWRb

[9] M. Pechyonkin, "Understanding hinton's capsule networks. part ii: How capsules work." November 2017.

[10] D. H. Ballard and C. M. Brown, "Computer vision. englewood cliffs," *J: Prentice Hall*, 1982.

[11] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[12] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[13] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," 2016, mIT Press. [Online]. Available: http://www.deeplearningbook.org

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1097–1105.

[15] M. D. Zeiler, G. W. Taylor, and R. Fergus, "Adaptive deconvolutional networks for mid and high level feature learning," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2018–2025.

[16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[18] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," *Computer Science Review*, vol. 37, p. 100270, 2020.

[19] A. Ramcharan, K. Baranowski, P. McCloskey, B. Ahmed, J. Legg, and D. P. Hughes, "Deep learning for image-based cassava disease detection,"

Frontiers in Plant Science, vol. 8, p. 1852, 2017. [Online]. Available: https://www.frontiersin.org/article/10.3389/fpls.2017.01852

[20] V. Miele, G. Dussert, B. Spataro, S. Chamaillé-Jammes, D. Allainé, and C. Bonenfant, "Revisiting animal photo-identification using deep metric learning and network analysis," *Methods in Ecology and Evolution*, vol. 12, no. 5, pp. 863–873, 2021.

[21] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland *et al.*, "Improved protein structure prediction using potentials from deep learning," *Nature*, vol. 577, no. 7792, pp. 706–710, 2020.

[22] C. Quintero, F. Merchán, A. Cornejo, and J. S. Galán, "Uso de redes neuronales convolucionales para el reconocimiento automático de imágenes de macroinvertebrados para el biomonitoreo participativo," *KnE Engineering*, pp. 585–596, 2018.

[23] A. Luckow, M. Cook, N. Ashcraft, E. Weill, E. Djerekarov, and B. Vorster, "Deep learning in the automotive industry: Applications and tools," in *Big Data, 2016 IEEE International Conference*. IEEE, 2016, pp. 3759–3768.

[24] J. P. Cohen, P. Morrison, and L. Dao, "Covid-19 image data collection," *arXiv 2003.11597*, 2020. [Online]. Available: https://github.com/ieee8023/covid-chestxray-dataset

[25] A. Mobiny and H. Van Nguyen, "Fast capsnet for lung cancer screening," in *Medical Image Computing and Computer Assisted Intervention–MICCAI 2018: 21st International Conference, Granada, Spain, September 16-20, 2018, Proceedings, Part II 11*. Springer, 2018, pp. 741–749.

[26] P. Afshar, A. Mohammadi, and K. N. Plataniotis, "Brain tumor type classification via capsule networks," in *2018 25th IEEE international conference on image processing (ICIP)*. IEEE, 2018, pp. 3129–3133.

[27] ——, "Brain tumor type classification via capsule networks," *CoRR*, vol. abs/1802.10200, 2018. [Online]. Available: http://arxiv.org/abs/1802.10200

[28] K. Kruthika, H. Maheshappa, A. D. N. Initiative *et al.*, "Cbir system using capsule networks and 3d cnn for alzheimer's disease diagnosis," *Informatics in Medicine Unlocked*, vol. 14, pp. 59–68, 2019.

[29] H. Dammu, T. Ren, and T. Q. Duong, "Deep learning prediction of pathological complete response, residual cancer burden, and progression-free survival in breast cancer patients," *Plos one*, vol. 18, no. 1, p. e0280148, 2023.

[30] M. Nasser and U. K. Yusof, "Deep learning based methods for breast cancer diagnosis: A systematic review and future direction," *Diagnostics*, vol. 13, no. 1, p. 161, 2023.

[31] M. Stumpe and C. Mermel, "Applying deep learning to metastatic breast cancer detection," October 2018, google AI.

[32] Y. Gurovich, Y. Hanani, O. Bar, G. Nadav, N. Fleischer, D. Gelbman, L. Basel-Salmon, P. M. Krawitz, S. B. Kamphausen, M. Zenker *et al.*, "Identifying facial phenotypes of genetic disorders using deep learning," *Nature medicine*, vol. 25, no. 1, pp. 60–64, 2019.

[33] Y. Bengio, Y. Lecun, and G. Hinton, "Deep learning for ai," *Communications of the ACM*, vol. 64, no. 7, pp. 58–65, 2021.

[34] DARPA, "Learning with less labels (LwLL)," Defense Advanced Research Projects Agency, Tech. Rep., 2018.

[35] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," *arXiv preprint arXiv:1506.06579*, 2015.

[36] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[37] D. H. HUBEL and T. N. WIESEL, "Receptive fields of single neurones in the cat's striate cortex," *Journal of physiology*, 1959.

[38] Y. LeCun, C. Cortes, and C. J. B. Yann, "The mnist database of handwritten digits," Internet, 1998, available: http://yann.lecun.com/exdb/mnist.

[39] S. Yang, F. Lee, R. Miao, J. Cai, L. Chen, W. Yao, K. Kotani, and Q. Chen, "Rs-capsnet: An advanced capsule network," *IEEE Access*, vol. 8, pp. 85 007–85 018, 2020.

[40] W. Huang and F. Zhou, "Da-capsnet: dual attention mechanism capsule network," *Scientific Reports*, vol. 10, no. 1, pp. 1–13, 2020.

[41] D. M. Sundaram and A. Loganathan, "Fsscaps-detcountnet: fuzzy soft sets and capsnet-based detection and counting network for monitoring animals from aerial images," *Journal of Applied Remote Sensing*, vol. 14, no. 2, p. 026521, 2020.

[42] B. Jia and Q. Huang, "De-capsnet: A diverse enhanced capsule network with disperse dynamic routing," *Applied Sciences*, vol. 10, no. 3, p. 884, 2020.

[43] A. Wang, M. Wang, H. Wu, K. Jiang, and Y. Iwahori, "A novel lidar data classification algorithm combined capsnet with resnet," *Sensors*, vol. 20, no. 4, p. 1151, 2020.

[44] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[45] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.

[46] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[47] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[48] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, and C. Igel, "Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark," in *International Joint Conference on Neural Networks*, no. 1288, 2013.

[49] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba, "Sun database: Large-scale scene recognition from abbey to zoo," in *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*. IEEE, 2010, pp. 3485–3492.

[50] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3D object representations for fine-grained categorization," in *Computer Vision Workshops (ICCVW), 2013 IEEE International Conference on*. IEEE, 2013, pp. 554–561.

[51] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[52] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[53] A. Barbu, D. Mayo, J. Alverio, W. Luo, C. Wang, D. Gutfreund, J. Tenenbaum, and B. Katz, "Objectnet: A large-scale bias-controlled dataset for pushing the limits of object recognition models," 2019.

[54] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[55] L. G. Shapiro and G. C. Stockman, *Computer Vision*, N. Jersey, Ed. Prentice-Hall, 2001, 279–325.

[56] L. Wang, Z. Q. Lin, and A. Wong, "Covid-net: a tailored deep convolutional neural network design for detection of covid-19 cases from chest x-ray images," *Scientific Reports*, vol. 10, no. 1, p. 19549, Nov 2020. [Online]. Available: https://doi.org/10.1038/s41598-020-76550-z

[57] T. Rahman, A. Khandakar, Y. Qiblawey, A. Tahir, S. Kiranyaz, S. B. A. Kashem, M. T. Islam, S. Al Maadeed, S. M. Zughaier, M. S. Khan *et al.*, "Exploring the effect of image enhancement techniques on covid-19 detection using chest x-ray images," *Computers in biology and medicine*, vol. 132, p. 104319, 2021.

[58] M. E. Chowdhury, T. Rahman, A. Khandakar, R. Mazhar, M. A. Kadir, Z. B. Mahbub, K. R. Islam, M. S. Khan, A. Iqbal, N. Al Emadi *et al.*, "Can ai help in screening viral and covid-19 pneumonia?" *IEEE Access*, vol. 8, pp. 132 665–132 676, 2020.

[59] X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, and R. M. Summers, "Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2097–2106.

[60] E. B. Tsai, S. Simpson, M. P. Lungren, M. Hershman, L. Roshkovan, E. Colak, B. J. Erickson, G. Shih, A. Stein, J. Kalpathy-Cramer *et al.*, "The rsna international covid-19 open radiology database (ricord)," *Radiology*, vol. 299, no. 1, pp. E204–E213, 2021.

[61] P. W. McCulloch Warren, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec 1943. [Online]. Available: https://doi.org/10.1007/BF02478259

[62] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[63] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[64] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the trade*. Springer, 2002, pp. 55–69.

[65] H. Wang and B. Raj, "On the origin of deep learning," *arXiv preprint arXiv:1702.07800*, 2017.

[66] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, p. 533, 1986.

[67] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[68] Xiangyang Liu and Hua Gu, "Hyperbolic tangent function based two layers structure neural network," in *Proceedings of 2011 International Conference on Electronics and Optoelectronics*, vol. 4, 2011, pp. V4–376–V4–379.

[69] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[70] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1, 2013, p. 3.

[71] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[72] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[73] M. K. Patrick, A. F. Adekoya, A. A. Mighty, and B. Y. Edward, "Capsule networks–a survey," *Journal of King Saud University-computer and information sciences*, vol. 34, no. 1, pp. 1295–1310, 2022.

[74] M. J. Alam, R. B. Rashid, S. A. Fattah, and M. Saquib, "Rat-capsnet: A deep learning network utilizing attention and regional information for abnormality detection in wireless capsule endoscopy," *IEEE Journal of Translational Engineering in Health and Medicine*, vol. 10, pp. 1–8, 2022.

[75] M. Khodadadzadeh, X. Ding, P. Chaurasia, and D. Coyle, "A hybrid capsule network for hyperspectral image classification," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 14, pp. 11 824–11 839, 2021.

[76] W. Wang, F. Lee, S. Yang, and Q. Chen, "An improved capsule network based on capsule filter routing," *IEEE Access*, vol. 9, pp. 109 374–109 383, 2021.

[77] N. A. K. Steur and F. Schwenker, "Next-generation neural networks: Capsule networks with routing-by-agreement for text classification," *IEEE Access*, vol. 9, pp. 125 269–125 299, 2021.

[78] R. Ma, T. Yu, X. Zhong, Z. L. Yu, Y. Li, and Z. Gu, "Capsule network for erp detection in brain-computer interface," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 29, pp. 718–730, 2021.

[79] L. Luo, L. Zhang, M. Wang, Z. Liu, X. Liu, R. He, and Y. Jin, "A system for the detection of polyphonic sound on a university campus based on capsnet-rnn," *IEEE Access*, vol. 9, pp. 147 900–147 913, 2021.

[80] H. Fang, J.-Q. Liu, K. Xie, P. Wu, X.-Y. Zhang, C. Wen, and J.-B. He, "Mr-capsnet: A deep learning algorithm for image-based head pose estimation on capsnet," *IEEE Access*, vol. 9, pp. 141 245–141 257, 2021.

[81] Z. Hu*, Z. Yang*, X. Hu, and R. Nevaita, "SimPLE: Similar Pseudo Label Exploitation for Semi-Supervised Classification," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021. [Online]. Available: https://arxiv.org/abs/2103.16725

[82] K. Ren, T. Zheng, Z. Qin, and X. Liu, "Adversarial attacks and defenses in deep learning," *Engineering*, 2020.

[83] W. Brendel, J. Rauber, A. Kurakin, N. Papernot, B. Veliqi, S. P. Mohanty, F. Laurent, M. Salathé, M. Bethge, Y. Yu *et al.*, "Adversarial vision challenge," in *The NeurIPS'18 Competition*. Springer, 2020, pp. 129–153.

[84] A. Jalal, A. Salman, A. Mian, M. Shortis, and F. Shafait, "Fish detection and species classification in underwater environments using deep learning with temporal information," *Ecological Informatics*, vol. 57, p. 101088, 2020.

[85] M. K. Patrick, A. F. Adekoya, A. A. Mighty, and B. Y. Edward, "Capsule networks–a survey," *Journal of King Saud University-Computer and Information Sciences*, 2019.

[86] V. Jayasundara, S. Jayasekara, H. Jayasekara, J. Rajasegaran, S. Seneviratne, and R. Rodrigo, "Textcaps: Handwritten character recognition with very small datasets," in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2019, pp. 254–262.

[87] V. M. d. Rosario, E. Borin, and M. Breternitz Jr, "The multi-lane capsule network (mlcn)," *arXiv preprint arXiv:1902.08431*, 2019.

[88] K. Kruthika, H. Maheshappa, A. D. N. Initiative *et al.*, "Cbir system using capsule networks and 3d cnn for alzheimer's disease diagnosis," *Informatics in Medicine Unlocked*, vol. 14, pp. 59–68, 2019.

[89] J. Rajasegaran, V. Jayasundara, S. Jayasekara, H. Jayasekara, S. Seneviratne, and R. Rodrigo, "Deepcaps: Going deeper with capsule networks," *CoRR*, vol. abs/1904.09546, 2019. [Online]. Available: http://arxiv.org/abs/1904.09546

[90] T. Hahn, M. Pyeon, and G. Kim, "Self-routing capsule networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 7656–7665.

[91] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 5, pp. 828–841, 2019.

[92] T. Zheng, C. Chen, and K. Ren, "Distributionally adversarial attack," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 2253–2260.

[93] T. B. Brown, N. Carlini, C. Zhang, C. Olsson, P. Christiano, and I. Goodfellow, "Unrestricted adversarial examples," *arXiv preprint arXiv:1809.08352*, 2018.

[94] A. R. Kosiorek, S. Sabour, Y. W. Teh, and G. E. Hinton, "Stacked capsule autoencoders," *arXiv preprint arXiv:1906.06818*, 2019.

[95] M. Amer and T. Maul, "Path capsule networks," *CoRR*, vol. abs/1902.03760, 2019. [Online]. Available: http://arxiv.org/abs/1902.03760

[96] N. Carlini and D. Wagner, "Audio adversarial examples: Targeted attacks on speech-to-text," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 1–7.

[97] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long, "Technical report on the cleverhans v2.1.0 adversarial examples library," *arXiv preprint arXiv:1610.00768*, 2018.

[98] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 9185–9193.

[99] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating adversarial examples with adversarial networks," *arXiv preprint arXiv:1801.02610*, 2018.

[100] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer, "Adversarial patch," *arXiv preprint arXiv:1712.09665*, 2017.

[101] N. Frosst, S. Sabour, and G. Hinton, "Darccc: Detecting adversaries by reconstruction from class conditional capsules," *arXiv preprint arXiv:1811.06969*, 2018.

[102] R. LaLonde and U. Bagci, "Capsules for object segmentation," *arXiv preprint arXiv:1804.04241*, 2018.

[103] R. Mukhometzianov and J. Carrillo, "Capsnet comparative performance evaluation for image classification," *arXiv preprint arXiv:1805.11195*, 2018.

[104] S. S. R. Phaye, A. Sikka, A. Dhall, and D. Bathula, "Dense and diverse capsule networks: Making the capsules learn better," *arXiv preprint arXiv:1805.04001*, 2018.

[105] J. O. Neill, "Siamese capsule networks," *arXiv preprint arXiv:1805.07242*, 2018.

[106] A. Raghunathan, J. Steinhardt, and P. Liang, "Certified defenses against adversarial examples," *arXiv preprint arXiv:1801.09344*, 2018.

[107] E. Wong and Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," in *International Conference on Machine Learning*. PMLR, 2018, pp. 5286–5295.

[108] Y. Guo, C. Zhang, C. Zhang, and Y. Chen, "Sparse dnns with improved adversarial robustness," *Advances in neural information processing systems*, vol. 31, pp. 242–251, 2018.

[109] Y. Wang, S. Jha, and K. Chaudhuri, "Analyzing the robustness of nearest neighbors to adversarial examples," in *International Conference on Machine Learning*. PMLR, 2018, pp. 5133–5142.

[110] X. Liu, Y. Li, C. Wu, and C.-J. Hsieh, "Adv-bnn: Improved adversarial defense through robust bayesian neural network," *arXiv preprint arXiv:1810.01279*, 2018.

[111] Q. Lei, L. Wu, P. Chen, A. G. Dimakis, I. S. Dhillon, and M. Witbrock, "Discrete attacks and submodular optimization with applications to text classification," *CoRR*, vol. abs/1812.00151, 2018. [Online]. Available: http://arxiv.org/abs/1812.00151

[112] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 1625–1634.

[113] S. A. Siddiqui, A. Salman, M. I. Malik, F. Shafait, A. Mian, M. R. Shortis, and E. S. Harvey, "Automatic fish species classification in underwater videos: exploiting pre-trained deep neural network models to compensate for limited labelled data," *ICES Journal of Marine Science*, vol. 75, no. 1, pp. 374–389, 2018.

[114] A. Jaiswal, W. AbdAlmageed, Y. Wu, and P. Natarajan, "Capsulegan: Generative adversarial capsule network," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 0–0.

[115] C. Xiang, L. Zhang, Y. Tang, W. Zou, and C. Xu, "Ms-capsnet: A novel multi-scale capsule network," *IEEE Signal Processing Letters*, vol. 25, no. 12, pp. 1850–1854, 2018.

[116] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. IEEE, 2017, pp. 39–57.

[117] E. Xi, S. Bing, and Y. Jin, "Capsule network performance on complex data," *arXiv preprint arXiv:1712.03480*, 2017.

[118] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[119] A. Sinha, H. Namkoong, R. Volpi, and J. Duchi, "Certifying some distributional robustness with principled adversarial training," *arXiv preprint arXiv:1710.10571*, 2017.

[120] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh, "Ead: elastic-net attacks to deep neural networks via adversarial examples," *arXiv preprint arXiv:1709.04114*, 2017.

[121] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against deep learning systems using adversarial examples," *CoRR*, vol. abs/1602.02697, 2016. [Online]. Available: http://arxiv.org/abs/1602.02697

[122] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[123] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2016, pp. 372–387.

[124] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.

[125] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 582–597.

[126] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: A simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[127] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[128] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[129] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[130] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.

[131] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularization of deep convolutional neural networks," *arXiv preprint arXiv:1301.3557*, 2013.

[132] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[133] G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *International Conference on Artificial Neural Networks*. Springer, 2011, pp. 44–51.

[134] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[135] S. Jie, Z. Xiaoteng, D. Zhengyan, Z. Yixin, C. Yanjun, Z. Jianying, W. Wenfei, M. Lin, and H. Chuanping, "Good practices for deep feature fusion," in *European Conference on Computer Vision*, ECCV, Ed., 2016.

[136] A. Serban, E. Poll, and J. Visser, "Adversarial examples on object recognition: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[137] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *Ieee Access*, vol. 6, pp. 14 410–14 430, 2018.

[138] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing robust adversarial examples," in *International conference on machine learning*. PMLR, 2018, pp. 284–293.

[139] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[140] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1765–1773.

[141] S. Sarkar, A. Bansal, U. Mahbub, and R. Chellappa, "Upset and angri: Breaking high performance image classifiers," *arXiv preprint arXiv:1707.01159*, 2017.

[142] C. Kanbak, S.-M. Moosavi-Dezfooli, and P. Frossard, "Geometric robustness of deep networks: analysis and improvement," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4441–4449.

[143] G. K. Dziugaite, Z. Ghahramani, and D. M. Roy, "A study of the effect of jpg compression on adversarial images," *arXiv preprint arXiv:1608.00853*, 2016.

[144] C. Guo, M. Rana, M. Cisse, and L. Van Der Maaten, "Countering adversarial images using input transformations," *arXiv preprint arXiv:1711.00117*, 2017.

[145] Y. Luo, X. Boix, G. Roig, T. Poggio, and Q. Zhao, "Foveation-based mechanisms alleviate adversarial examples," *arXiv preprint arXiv:1511.06292*, 2015.

[146] C. Xie, J. Wang, Z. Zhang, Y. Zhou, L. Xie, and A. Yuille, "Adversarial examples for semantic segmentation and object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1369–1378.

[147] Y. Song, T. Kim, S. Nowozin, S. Ermon, and N. Kushman, "Pixeldefend: Leveraging generative models to understand and defend against adversarial examples," *arXiv preprint arXiv:1710.10766*, 2017.

[148] J. Lu, T. Issaranon, and D. Forsyth, "Safetynet: Detecting and rejecting adversarial examples robustly," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 446–454.

[149] B. Liang, H. Li, M. Su, X. Li, W. Shi, and X. Wang, "Detecting adversarial image examples in deep neural networks with adaptive noise reduction," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 72–85, 2018.

[150] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," *arXiv preprint arXiv:1412.5068*, 2014.

[151] A. Sinha, Z. Chen, V. Badrinarayanan, and A. Rabinovich, "Gradient adversarial training of neural networks," *arXiv preprint arXiv:1806.08028*, 2018.

[152] C. Lyu, K. Huang, and H.-N. Liang, "A unified gradient regularization family for adversarial examples," in *2015 IEEE international conference on data mining*. IEEE, 2015, pp. 301–309.

[153] U. Shaham, Y. Yamada, and S. Negahban, "Understanding adversarial training: Increasing local stability of neural nets through robust optimization," *arXiv preprint arXiv:1511.05432*, 2015.

[154] G. S. Dhillon, K. Azizzadenesheli, Z. C. Lipton, J. Bernstein, J. Kossaifi, A. Khanna, and A. Anandkumar, "Stochastic activation pruning for robust adversarial defense," *arXiv preprint arXiv:1803.01442*, 2018.

[155] M. Cisse, Y. Adi, N. Neverova, and J. Keshet, "Houdini: Fooling deep structured prediction models," *arXiv preprint arXiv:1707.05373*, 2017.

[156] J. Gao, B. Wang, Z. Lin, W. Xu, and Y. Qi, "Deepcloak: Masking deep neural network models for robustness against adversarial samples," *arXiv preprint arXiv:1702.06763*, 2017.

[157] N. Akhtar, J. Liu, and A. Mian, "Defense against universal adversarial perturbations," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 3389–3398.

[158] H. Lee, S. Han, and J. Lee, "Generative adversarial trainer: Defense to adversarial perturbations with gan," *arXiv preprint arXiv:1705.03387*, 2017.

[159] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017.

[160] D. Meng and H. Chen, "Magnet: a two-pronged defense against adversarial examples," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 135–147.

[161] A. Kurakin, I. Goodfellow, S. Bengio, Y. Dong, F. Liao, M. Liang, T. Pang, J. Zhu, X. Hu, C. Xie *et al.*, "Adversarial attacks and defences competition," in *The NIPS'17 Competition: Building Intelligent Systems*. Springer, 2018, pp. 195–231.

[162] G. Hinton, "How to represent part-whole hierarchies in a neural network," *arXiv preprint arXiv:2102.12627*, 2021.

[163] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 2921–2926.

[164] H. Xiang, Y.-S. Huang, C.-H. Lee, T.-Y. C. Chien, C.-K. Lee, L. Liu, A. Li, X. Lin, and R.-F. Chang, "3-d res-capsnet convolutional neural network on automated breast ultrasound tumor diagnosis," *European Journal of Radiology*, vol. 138, p. 109608, 2021.

[165] A. Mittal, D. Kumar, M. Mittal, T. Saba, I. Abunadi, A. Rehman, and S. Roy, "Detecting pneumonia using convolutions and dynamic capsule routing for chest x-ray images," *Sensors*, vol. 20, no. 4, p. 1068, 2020.

[166] M. Khanna, A. Agarwal, L. K. Singh, S. Thawkar, A. Khanna, and D. Gupta, "Radiologist-level two novel and robust automated computer-aided prediction models for early detection of covid-19 infection from chest x-ray images," *Arabian Journal for Science and Engineering*, pp. 1–33, 2021.

[167] P. Afshar, S. Heidarian, F. Naderkhani, A. Oikonomou, K. N. Plataniotis, and A. Mohammadi, "Covid-caps: A capsule network-based framework for identification of covid-19 cases from x-ray images," *Pattern Recognition Letters*, vol. 138, pp. 638–643, 2020.

[168] S. Toraman, T. B. Alakus, and I. Turkoglu, "Convolutional capsnet: A novel artificial neural network approach to detect covid-19 disease from x-ray images using capsule networks," *Chaos, Solitons & Fractals*, vol. 140, p. 110122, 2020.

[169] D. Filippas, C. Nicopoulos, and G. Dimitrakopoulos, "Streaming dilated convolution engine," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[170] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *arXiv preprint arXiv:1511.07122*, 2015.

[171] "TensorFlow Datasets, a collection of ready-to-use datasets," https://www.tensorflow.org/datasets.

[172] E. Xi, S. Bing, and Y. Jin, "Capsule network performance on complex data," *arXiv preprint arXiv:1712.03480*, 2017.

[173] M. U. Ali, K. D. Kallu, H. Masood, U. Tahir, C. V. Gopi, A. Zafar, S. W. Lee *et al.*, "A cnn-based chest infection diagnostic model: A multistage multiclass isolated and developed transfer learning framework," *International Journal of Intelligent Systems*, vol. 2023, 2023.

[174] R. Sarki, K. Ahmed, H. Wang, Y. Zhang, and K. Wang, "Automated detection of covid-19 through convolutional neural network using chest x-ray images," *Plos one*, vol. 17, no. 1, p. e0262052, 2022.

Appendix A

Programming Codes

A.1 MNIST Program

```
1
2 #se cargan las librerias necesarias, Tensorflow ya esta con keras
3 import numpy as np
4 import capslayersTF2
5 import csv
6 import os
7 os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
8 os.environ["CUDA_VISIBLE_DEVICES"]="1"
9 import argparse
10 import tensorflow as tf
11 from tensorflow.keras import layers, models, optimizers, callbacks
12 import tensorflow.keras.backend as K
13 from tensorflow.keras.utils import to_categorical
14 from tensorflow.keras.preprocessing.image import ImageDataGenerator
15 from utils import combine_images
16 from PIL import Image
17 from capslayersTF2 import CapsuleLayer, PrimaryCaps, Length, Mask
18 from matplotlib import pyplot as plt
19 import pandas as pd
```

```
import seaborn as sn
from matplotlib import pyplot as plt
from sklearn.metrics import confusion_matrix
from pretty_confusion_matrix import pp_matrix

K.set_image_data_format('channels_last')    #me aseguro que se utilice el
    vector de forma [width height channels]


# Armo la arquitectura de CapsNet

def CapsNet(input_shape, n_class , routings, batch_size):
    """
    A Capsule Network on MNIST.
    :param input_shape: data shape, 3d, [width, height, channels]
    :param n_class: number of classes
    :param routings: number of routing iterations
    :param batch_size: size of batch
    :return: Two Keras Models, the first one used for training, and the
        second one for evaluation.
            `eval_model` can also be used for training.
    """

    x = tf.keras.Input(shape=input_shape, batch_size=batch_size)

     # Layer 1: Just a conventional Conv2D layer
    #conv1 = tf.keras.layers.Conv2D(256, 9, strides=(1, 1), padding='valid
        ',dilation_rate=(2, 2), activation='relu', name='conv1')(x)

    #version 3 de capsnets
    conv1 = tf.keras.layers.Conv2D(64, 3, strides=(1, 1), padding='valid',
        dilation_rate=(1, 1), activation='relu', name='conv1')(x)
    conv2 = tf.keras.layers.Conv2D(128, 3, strides=(1, 1), padding='valid',
        dilation_rate=(2, 2), activation='relu', name='conv2')(conv1)
    conv3 = tf.keras.layers.Conv2D(256, 3, strides=(1, 1), padding='valid',
        dilation_rate=(4, 4), activation='relu', name='conv3')(conv2)
```

```
50
51    # Layer 2: Conv2D layer with `squash` activation, then reshape to [None
          , num_capsule, dim_capsule]
52    primarycaps = PrimaryCaps(conv3, dim_capsule=8, n_channels=32,
          kernel_size=9, strides=2, padding='valid')
53
54    # Layer 3: Capsule layer. Routing algorithm works here.
55    digitcaps = CapsuleLayer(num_capsule=n_class, dim_capsule=16, routings=
          routings, name='digitcaps')(primarycaps)
56
57    # Layer 4: This is an auxiliary layer to replace each capsule with its
          length. Just to match the true label's shape.
58    # If using tensorflow, this will not be necessary. :)
59    out_caps = Length(name='capsnet')(digitcaps)
60
61    # Decoder network.
62    y = tf.keras.Input(shape=(n_class,))
63    masked_by_y = Mask()([digitcaps, y])  # The true label is used to mask
          the output of capsule layer. For training
64    masked = Mask()(digitcaps)  # Mask using the capsule with maximal
          length. For prediction
65
66    #Construimos un modelo llamado decoder para entrenar y la predecir
67    decoder=tf.keras.Sequential(name='decoder')
68    decoder.add(tf.keras.layers.Dense(512, activation='relu', input_dim=16*
          n_class))
69    decoder.add(tf.keras.layers.Dense(1024,activation='relu'))
70    decoder.add(tf.keras.layers.Dense(np.prod(input_shape), activation='
          sigmoid'))
71    decoder.add(tf.keras.layers.Reshape(target_shape=input_shape, name='
          out_recon'))
72
73    #Modelo para entrenar y evaluar (hacer las predicciones)
74
75    train_model = tf.keras.Model([x,y], [out_caps, decoder(masked_by_y)])
76    eval_model = tf.keras.Model(x, [out_caps, decoder(masked)])
```

```
77
78      #manipular el modelo
79      #noise = tf.keras.Input(shape=(n_class,16))
80      #noised_digitcaps = tf.keras.layers.Add()([digitcaps, noise])
81      #masked_noised_y = Mask()([noised_digitcaps,y])
82      #manipulate_model = tf.keras.Model([x, y, noise], decoder(
            masked_noised_y))
83
84      noise = layers.Input(shape=(n_class, 16))
85      noised_digitcaps = layers.Add()([digitcaps, noise])
86      masked_noised_y = Mask()([noised_digitcaps, y])
87      manipulate_model = models.Model([x, y, noise], decoder(masked_noised_y)
            )
88
89
90      return train_model, eval_model, manipulate_model
91
92
93  def margin_loss(y_true, y_pred):
94      """
95      Margin loss for Eq.(4). When y_true[i, :] contains not just one `1`,
            this loss should work too. Not test it.
96      :param y_true: [None, n_classes]
97      :param y_pred: [None, num_capsule]
98      :return: a scalar loss value.
99      """
100     # return tf.reduce_mean(tf.square(y_pred))
101
102     L = y_true * tf.square(tf.maximum(0., 0.9 - y_pred)) + \
103         0.5 * (1 - y_true) * tf.square(tf.maximum(0., y_pred - 0.1))
104
105     return tf.math.reduce_mean(tf.math.reduce_sum(L, 1))
106
107
108 def train(model,  # type: models.Model
109           data, args):
```

```
110     """
111     Training a CapsuleNet
112     :param model: the CapsuleNet model
113     :param data: a tuple containing training and testing data, like `((
            x_train, y_train), (x_test, y_test))`
114     :param args: arguments
115     :return: The trained model
116     """
117     # unpacking the data
118     (x_train, y_train), (x_test, y_test) = data
119
120     log = tf.keras.callbacks.CSVLogger(args.save_dir + '/log.csv')  #
            Callback that streams epoch results to a CSV file.
121     checkpoint = tf.keras.callbacks.ModelCheckpoint(args.save_dir + '/
            weights-{epoch:02d}.h5', monitor='val_capsnet_acc',
122                                                 save_best_only=True,
                                                        save_weights_only=True,
                                                        verbose=1)
123     lr_decay = tf.keras.callbacks.LearningRateScheduler(schedule=lambda
            epoch: args.lr * (args.lr_decay ** epoch))
124
125     # compile the model
126     model.compile(optimizer=optimizers.Adam(lr=args.lr),
127                   loss=[margin_loss, 'mse'],
128                   loss_weights=[1., args.lam_recon],
129                   metrics={'capsnet': 'accuracy'})
130
131      # Begin: Training with data augmentation
             ---------------------------------------------------------------------#

132     def train_generator(x, y, batch_size, shift_fraction=0.1):
133         train_datagen = ImageDataGenerator(width_shift_range=shift_fraction
                ,
134                                                 height_shift_range=
                                                        shift_fraction)  # shift up
                                                        to 2 pixel for MNIST
```

```
135         generator = train_datagen.flow(x, y, batch_size=batch_size)
136         while 1:
137             x_batch, y_batch = generator.next()
138             yield (x_batch, y_batch), (y_batch, x_batch)
139 
140     # Training with data augmentation. If shift_fraction=0., no
            augmentation.
141     model.fit(train_generator(x_train, y_train, args.batch_size, args.
            shift_fraction),
142               steps_per_epoch=int(y_train.shape[0] / args.batch_size),
143               epochs=args.epochs,
144               validation_data=((x_test, y_test), (y_test, x_test)),
                      batch_size=args.batch_size,
145               callbacks=[log, checkpoint, lr_decay])
146     # End: Training with data augmentation
            -----------------------------------------------------------------------#


147 
148     model.save_weights(args.save_dir + '/trained_model.h5')
149     print('Trained model saved to \'%s/trained_model.h5\'' % args.save_dir)
150 
151     from utils import plot_log
152     plot_log(args.save_dir + '/log.csv', show=True)
153 
154     return model
155 
156 
157 def test(model, data, args):
158     x_test, y_test = data
159     y_pred, x_recon = model.predict(x_test, batch_size=100)
160 
161     cm= confusion_matrix((np.argmax(y_pred, axis=1)),(np.argmax(y_test,
            axis=1)) )
162     print(cm)
163 
164     print('-' * 30 + 'Begin: test' + '-' * 30)
```

```
165     print('Test acc:', np.sum(np.argmax(y_pred, 1) == np.argmax(y_test, 1))
            / y_test.shape[0])
166
167     img = combine_images(np.concatenate([x_test[:50], x_recon[:50]]))
168     image = img * 255
169     Image.fromarray(image.astype(np.uint8)).save(args.save_dir + "/
            real_and_recon.png")
170     print()
171     print('Reconstructed images are saved to %s/real_and_recon.png' % args.
            save_dir)
172     print('-' * 30 + 'End: test' + '-' * 30)
173     plt.imshow(plt.imread(args.save_dir + "/real_and_recon.png"))
174     plt.show()
175
176
177 def manipulate_latent(model, data, args):
178     print('-' * 30 + 'Begin: manipulate' + '-' * 30)
179     x_test, y_test = data
180     index = np.argmax(y_test, 1) == args.digit
181     number = np.random.randint(low=0, high=sum(index) - 1)
182     x, y = x_test[index][number], y_test[index][number]
183     x, y = np.expand_dims(x, 0), np.expand_dims(y, 0)
184     noise = np.zeros([1, 10, 16])
185     x_recons = []
186     for dim in range(16):
187         for r in [-0.25, -0.2, -0.15, -0.1, -0.05, 0, 0.05, 0.1, 0.15, 0.2,
                0.25]:
188             tmp = np.copy(noise)
189             tmp[:, :, dim] = r
190             x_recon = model.predict([x, y, tmp])
191             x_recons.append(x_recon)
192
193     x_recons = np.concatenate(x_recons)
194
195     img = combine_images(x_recons, height=16)
196     image = img * 255
```

```
197     Image.fromarray(image.astype(np.uint8)).save(args.save_dir + '/
            manipulate-%d.png' % args.digit)
198     print('manipulated result saved to %s/manipulate-%d.png' % (args.
            save_dir, args.digit))
199     print('-' * 30 + 'End: manipulate' + '-' * 30)
200
201
202 def load_mnist():
203     # the data, shuffled and split between train and test sets
204     (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
            load_data()
205
206     #para juntarlos xy_train=tf.keras.dataset.zip((x_train,y_train))
207
208     print(x_train.shape) #(60000,28,28)
209     print(y_train.shape) #(60000,)
210
211     print(x_test.shape)  #(10000,28,28)
212     print(y_test.shape)  #(10000,)
213
214     x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255.
215     x_test = x_test.reshape(-1, 28, 28, 1).astype('float32') / 255.
216     y_train = to_categorical(y_train.astype('float32'))
217     y_test = to_categorical(y_test.astype('float32'))
218
219     print(type(x_train)) #(60000,28,28,1)
220     print(x_test.shape)  #(10000,28,28,1)
221
222     print(y_train.shape) #(60000,10)
223     print(y_test.shape)  #(10000,10)
224     return (x_train, y_train), (x_test, y_test)
225
226 if __name__ == "__main__":
227
228     # setting the hyper parameters
229     parser = argparse.ArgumentParser(description="Capsule Network on MNIST.
```

```
        ")
230     parser.add_argument('--epochs', default=5, type=int)
231     parser.add_argument('--batch_size', default=100, type=int)
232     parser.add_argument('--lr', default=0.001, type=float,
233                         help="Initial learning rate")
234     parser.add_argument('--lr_decay', default=0.9, type=float,
235                         help="The value multiplied by lr at each epoch. Set
                                a larger value for larger epochs")
236     parser.add_argument('--lam_recon', default=0.392, type=float,
237                         help="The coefficient for the loss of decoder")
238     parser.add_argument('-r', '--routings', default=3, type=int,
239                         help="Number of iterations used in routing
                                algorithm. should > 0")
240     parser.add_argument('--shift_fraction', default=0.1, type=float,
241                         help="Fraction of pixels to shift at most in each
                                direction.")
242     parser.add_argument('--debug', action='store_true',
243                         help="Save weights by TensorBoard")
244     parser.add_argument('--save_dir', default='./result')
245     parser.add_argument('-t', '--testing', action='store_true',
246                         help="Test the trained model on testing dataset")
247     parser.add_argument('--digit', default=5, type=int,
248                         help="Digit to manipulate")
249     parser.add_argument('-w', '--weights', default=None,
250                         help="The path of the saved weights. Should be
                                specified when testing")
251     args = parser.parse_args()
252     print(args)
253
254     if not os.path.exists(args.save_dir):
255         os.makedirs(args.save_dir)
256
257     ######  ESTE ES EL PROGRAMA PRINCIPAL  ######
258     # load data
259     (x_train, y_train), (x_test, y_test) = load_mnist()
260
```

```
261     # define model
262     model, eval_model, manipulate_model = CapsNet(input_shape=x_train.shape
            [1:],
263                                                   n_class=len(np.unique(np.
                                                          argmax(y_train, 1))),
264                                                   routings=args.routings,
265                                                   batch_size=args.
                                                          batch_size)
266     print(x_train.shape[1:])  #(28,28,1)
267     print(len(np.unique(np.argmax(y_train, 1)))) #(10)
268     print(args.routings)  #3
269     print(args.batch_size)   #100
270
271
272     model.summary()
273
274     #train(model=model, data=((x_train, y_train), (x_test, y_test)), args=
            args)
275     #model.load_weights(args.weights)
276     #manipulate_latent(manipulate_model, (x_test, y_test), args)
277     #test(model=eval_model, data=(x_test, y_test), args=args)
278
279     # train or test
280     if args.weights is not None:  # init the model weights with provided
            one
281         model.load_weights(args.weights)
282     if not args.testing:
283         train(model=model, data=((x_train, y_train), (x_test, y_test)),
                args=args)
284     else:  # as long as weights are given, will run testing
285         if args.weights is None:
286             print('No weights are provided. Will test using random
                    initialized weights.')
287         print("HOLA!!!")
288         #manipulate_latent(manipulate_model, (x_test, y_test), args)
289         print("HOLA ENFERMERA!!!")
```

```
290         test(model=eval_model, data=(x_test, y_test), args=args)
```

A.2 COVIDx Program

```
1 import numpy as np
2 import pandas as pd
3 import seaborn as sn
4 import capslayersTF2
5 import csv
6 import os
7 os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
8 os.environ["CUDA_VISIBLE_DEVICES"]="1"
9 import argparse
10 import tensorflow as tf
11 import tensorflow_datasets as tfds
12 from tensorflow.keras import layers, models, optimizers, callbacks
13 import tensorflow.keras.backend as K
14 from tensorflow.keras.utils import to_categorical
15 from tensorflow.keras.preprocessing.image import ImageDataGenerator
16 from utils import combine_images
17 from PIL import Image
18 from capslayersTF2 import CapsuleLayer, PrimaryCaps, Length, Mask
19 from matplotlib import pyplot as plt
20 from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
21 from sklearn.metrics import classification_report
22 from pretty_confusion_matrix import pp_matrix
23 K.set_image_data_format('channels_last')   #me aseguro que se utilice el
     vector de forma [width height channels]
24
25
26 def CapsNet(input_shape, n_class , routings, batch_size):
27     """
28     A Capsule Network on MNIST.
29     :param input_shape: data shape, 3d, [width, height, channels]
30     :param n_class: number of classes
```

```
31      :param routings: number of routing iterations
32      :param batch_size: size of batch
33      :return: Two Keras Models, the first one used for training, and the
            second one for evaluation.
34                `eval_model` can also be used for training.
35      """
36      x = tf.keras.Input(shape=input_shape, batch_size=batch_size, name='
            entrada')
37
38      conv1 = tf.keras.layers.Conv2D(128, 3, strides=(1, 1), padding='valid',
            dilation_rate=(8, 8), activation='relu', name='conv1')(x)
39      conv2 = tf.keras.layers.Conv2D(64, 3, strides=(2, 2), padding='valid',
            dilation_rate=(1, 1), activation='relu', name='conv2')(conv1)
40      conv3 = tf.keras.layers.Conv2D(128, 3, strides=(1, 1), padding='valid',
            dilation_rate=(4, 4), activation='relu', name='conv3')(conv2)
41      conv4 = tf.keras.layers.Conv2D(64, 3, strides=(2, 2), padding='valid',
            dilation_rate=(1, 1), activation='relu', name='conv4')(conv3)
42      conv5 = tf.keras.layers.Conv2D(64, 3, strides=(1, 1), padding='valid',
            dilation_rate=(2, 2), activation='relu', name='conv5')(conv4)
43      conv6 = tf.keras.layers.Conv2D(64, 3, strides=(2, 2), padding='valid',
            dilation_rate=(1, 1), activation='relu', name='conv6')(conv5)
44      conv7 = tf.keras.layers.Conv2D(64, 3, strides=(2, 2), padding='valid',
            dilation_rate=(1, 1), activation='relu', name='conv7')(conv6)
45
46      primarycaps = PrimaryCaps(conv7,dim_capsule=8, n_channels=32,
            kernel_size=9, strides=2, padding='valid')
47      digitcaps = CapsuleLayer(num_capsule=n_class, dim_capsule=16, routings=
            routings, name='digitcaps')(primarycaps)
48      out_caps = Length(name='capsnet')(digitcaps)
49
50      # Decoder network.
51      y = tf.keras.Input(shape=(n_class,))
52
53      masked_by_y = Mask()([digitcaps, y])
54      masked = Mask()(digitcaps)  # Mask using the capsule with maximal
            length. For prediction
```

```
55
56      #Construimos un modelo llamado decoder para entrenar y la predecir
57      decoder=tf.keras.Sequential(name='decoder')
58      decoder.add(tf.keras.layers.Dense(64, activation='relu', input_dim=16*
            n_class))    #aqui cambiar el dim capsule
59      decoder.add(tf.keras.layers.Dense(128,activation='relu'))
60      decoder.add(tf.keras.layers.Dense(128,activation='relu'))  #probar
            cambiar aqui a 256
61      decoder.add(tf.keras.layers.Dense(np.prod(input_shape), activation='
            sigmoid'))
62      decoder.add(tf.keras.layers.Reshape(target_shape=input_shape, name='
            out_recon'))
63
64      #Modelo para entrenar y evaluar (hacer las predicciones)
65      print('##########estructura del modelo#######')
66      train_model = tf.keras.Model([x,y], [out_caps, decoder(masked_by_y)])
67      eval_model = tf.keras.Model(x, [out_caps, decoder(masked)])
68      return train_model, eval_model
69
70  def margin_loss(y_true, y_pred):
71      """
72      Margin loss for Eq.(4). When y_true[i, :] contains not just one `1`,
            this loss should work too. Not test it.
73      :param y_true: [None, n_classes]
74      :param y_pred: [None, num_capsule]
75      :return: a scalar loss value.
76      """
77      L = y_true * tf.square(tf.maximum(0., 0.9 - y_pred)) + \
78          0.5 * (1 - y_true) * tf.square(tf.maximum(0., y_pred - 0.1))
79
80      return tf.math.reduce_mean(tf.math.reduce_sum(L, 1))
81
82
83  def preprocess_train(image,label):
84      grayscaled = tf.image.rgb_to_grayscale(image)
85      grayscaled = grayscaled / 255.
```

```
86      return (grayscaled, label), (label,grayscaled)  # > - aqui es ta el
            punto, lo ponemos como funcion de prepocesamiento!
87
88  def preprocess_test(image,label):
89      grayscaled = tf.image.rgb_to_grayscale(image)
90      grayscaled = grayscaled / 255.
91      return (grayscaled, label)#,(label,grayscaled)  #cehacr esta parte para
            la prediccion
92
93  def get_testing_dataset():
94      batch_size=32
95      root_dir = os.path.abspath('.')
96      data_dir = os.path.join(root_dir,'COVID dataset')
97      testing_data_dir = os.path.join(data_dir,'test')
98
99      test_ds = tf.keras.preprocessing.image_dataset_from_directory(
100         testing_data_dir,
101         labels='inferred',
102         seed=123,
103         #shuffle=False,  # False las pasa alfanumericamente, True es
               default revuelve las imagenes checar resultados por que varian
               las imagenes recosntruidas
104         image_size=(args.img_width, args.img_height),
105         #color_mode = 'grayscale',
106         batch_size=batch_size,
107         label_mode='categorical')
108
109     AUTOTUNE = tf.data.AUTOTUNE
110     #test_dataset = test_ds.map(preprocess_test, num_parallel_calls=
            AUTOTUNE
111     #                               ).shuffle(1000).take(batch_size).cache()
112     test_dataset = test_ds.map(preprocess_test)
113     return test_dataset
114
115
116 def get__training_dataset(batch_size=32):
```

```
117
118     root_dir = os.path.abspath('.')
119     data_dir = os.path.join(root_dir,'COVID dataset')
120     train_data_dir = os.path.join(data_dir,'train')
121
122     train_ds = tf.keras.preprocessing.image_dataset_from_directory(
123         train_data_dir,
124         validation_split=0.2,
125         subset="training",
126         seed=123,
127         labels='inferred',
128         image_size=(args.img_width,args.img_height),
129         #color_mode = 'grayscale',
130         batch_size=batch_size,
131         label_mode='categorical')
132
133     for xy,y in train_ds.take(1):
134         print('hola enfermera')
135         print(xy.shape)
136         print(y.shape)
137         break  # para mostrar solo las dimensiones de un lote
138
139     val_ds = tf.keras.preprocessing.image_dataset_from_directory(
140         train_data_dir,
141         validation_split=0.2,
142         subset="validation",
143         seed=123,
144         labels='inferred',
145         image_size=(args.img_width,args.img_height),
146         #color_mode = 'grayscale',
147         batch_size=batch_size,
148         label_mode='categorical')
149
150     for xy,y in val_ds.take(1):
151         print('hola enfermera2')
152         print(xy.shape)
```

```
153         print(y.shape)
154         break  # para mostrar solo las dimensiones de un lote
155
156     class_names = train_ds.class_names
157     print(class_names)
158
159     AUTOTUNE = tf.data.AUTOTUNE
160     train_dataset = train_ds.map(preprocess_train, num_parallel_calls=
            AUTOTUNE
161                                  ).shuffle(1000).take(batch_size).cache()
                                         .repeat()
162
163     val_dataset = val_ds.map(preprocess_train, num_parallel_calls=AUTOTUNE
164                                  ).shuffle(1000).take(batch_size).cache()
                                         .repeat()
165
166     return train_dataset, val_dataset
167
168 def graficar_resultados(history):
169     history_dict = history.history
170
171     accuracy_values=history_dict["capsnet_accuracy"]
172     val_accuracy_values=history_dict["val_capsnet_accuracy"]
173     epochs = range(1, len(accuracy_values)+1)
174     plt.plot(epochs,accuracy_values,"r", label="capsnet_accuracy")
175     plt.plot(epochs,val_accuracy_values,"b", label='
            validation_capsnet_accuracy')
176     plt.title("Training and validation accuracy")
177     plt.xlabel("epochs")
178     plt.ylabel("accuracy")
179     plt.legend()
180     plt.savefig(args.save_dir + '/capsnet_accuracy.png')
181
182     plt.clf()
183     loss_values=history_dict["loss"]
184     val_loss_values=history_dict["val_loss"]
```

```
185     epochs = range(1, len(loss_values)+1)
186     plt.plot(epochs,loss_values,"r", label="training loss")
187     plt.plot(epochs,val_loss_values,"b", label="validation loss")
188     plt.title("Training and validation loss")
189     plt.xlabel("epochs")
190     plt.ylabel("loss")
191     plt.legend()
192     plt.savefig(args.save_dir + '/lossvsval_loss.png')
193 
194     plt.clf()
195     capsnet_loss_values=history_dict["capsnet_loss"]
196     val_capsnet_loss_values=history_dict["val_capsnet_loss"]
197     epochs = range(1, len(capsnet_loss_values)+1)
198     plt.plot(epochs,capsnet_loss_values,"r", label="training_capsnet_loss")
199     plt.plot(epochs,val_capsnet_loss_values,"b", label="val_capsnet_loss")
200     plt.title("Training and validation capsnet loss")
201     plt.xlabel("epochs")
202     plt.ylabel("loss")
203     plt.legend()
204     plt.savefig(args.save_dir + '/capslossvscapsval_loss.png')
205 
206     plt.clf()
207     decoder_loss_values=history_dict["decoder_loss"]
208     val_decoder_loss_values=history_dict["val_decoder_loss"]
209     epochs = range(1, len(decoder_loss_values)+1)
210     plt.plot(epochs,decoder_loss_values,"r", label="training decoder loss")
211     plt.plot(epochs,val_decoder_loss_values,"b", label="validation decoder
            loss")
212     plt.title("Training and validation decoder loss")
213     plt.xlabel("epochs")
214     plt.ylabel("loss")
215     plt.legend()
216     plt.savefig(args.save_dir + '/decoderlossvsdecoderval_loss.png')
217 
218     plt.clf()
219     plt.plot(epochs,decoder_loss_values,"r", label="training_decoder_loss")
```

```
220     plt.plot(epochs,loss_values,"b", label="training loss")
221     plt.plot(epochs,capsnet_loss_values,"g", label="training_capsnet_loss")
222     plt.title("Training loss")
223     plt.xlabel("epochs")
224     plt.ylabel("loss")
225     plt.legend()
226     plt.savefig(args.save_dir + '/training_loss.png')
227
228
229
230
231 def train(model,
232           data,# type: models.Model
233            args):
234     """
235     Training a CapsuleNet
236     :param model: the CapsuleNet model
237     :param data: a tuple containing training and testing data, like `((
            x_train, y_train), (x_test, y_test))`
238     :param args: arguments
239     :return: The trained model
240     """
241
242     log = tf.keras.callbacks.CSVLogger(args.save_dir + '/log.csv')
243     checkpoint = tf.keras.callbacks.ModelCheckpoint(args.save_dir + '/
            weights-{epoch:02d}.h5', monitor='val_capsnet_acc',
244                                                 save_best_only=True,
                                                        save_weights_only=True,
                                                        verbose=1)
245     lr_decay = tf.keras.callbacks.LearningRateScheduler(schedule=lambda
            epoch: args.lr * (args.lr_decay ** epoch))
246
247     # compile the model
248     model.compile(optimizer=optimizers.Adam(learning_rate=args.lr),
249                   loss=[margin_loss, 'mae'],
250                   loss_weights=[1., args.lam_recon],
```

```
251                   metrics={'capsnet': 'accuracy'})
252
253     train_ds, val_ds = data
254     steps_per_epoch=np.ceil(12089 / args.batch_size)
255     print(steps_per_epoch)
256
257     history = model.fit(train_ds,
258               steps_per_epoch=steps_per_epoch,
259               epochs=args.epochs,
260               validation_data=val_ds,
261               validation_steps=(3022 // args.batch_size),
262               callbacks=[log, checkpoint, lr_decay])
263
264     graficar_resultados(history)
265     model.save_weights(args.save_dir + '/trained_model.h5')
266     print('Trained model saved to \'%s/trained_model.h5\'' % args.save_dir)
267
268     from utils import plot_log
269     plot_log(args.save_dir + '/log.csv', show=True)
270
271     return model
272
273
274 def test(model, args):      #antes (model, data ,args)
275     print('datos1')
276     test_ds = get_testing_dataset()
277
278     x_test, y_test = next(iter(test_ds))
279
280     for x,y in test_ds.as_numpy_iterator():
281         x_test = np.concatenate((x_test, x),axis=0)
282         y_test = np.concatenate((y_test, y),axis=0)
283
284     x_test = x_test[32:1600] #antes 32
285     y_test = y_test[32:1600]
286     print('datos3')
```

```
287     y_pred, x_recon = model.predict(
288                         #test_ds,
289                         x_test,
290                         batch_size=32,
291                         verbose="1",
292                         steps=49)
293 
294     print('datos4')
295 
296     print(x_recon.shape)
297     print(y_pred.shape)
298     print(x_test.shape)
299     print(y_test.shape)
300 
301     print('NUEVO!!!')
302     classes = ['covid','healthy','pneumonia']
303     print(f'Reporte de clasificaci n:')
304     print(classification_report(np.argmax(y_test, axis=1),
305                                 np.argmax(y_pred, axis=1),
306                                 target_names=classes))
307 
308     print('datos5')
309     cm= confusion_matrix((np.argmax(y_test, axis=1)),(np.argmax(y_pred,
            axis=1)) )
310     #print(np.argmax(y_test, axis=1))
311     #print(np.argmax(y_pred, axis=1))
312     #print(cm)
313     cm_display = ConfusionMatrixDisplay(cm,display_labels=classes).plot()
314     print(cm_display)
315     plt.savefig(args.save_dir + '/confusion matrix1')
316 
317     df_cm = pd.DataFrame(cm, index=classes, columns= classes)
318     print('confusion matrix')
319     print(df_cm)
320 
321     print('-' * 30 + 'Begin: test' + '-' * 30)
```

```
322     print('Test acc:', np.sum(np.argmax(y_pred, 1) == np.argmax(y_test, 1))
            / y_test.shape[0])
323
324     img = combine_images(np.concatenate([x_test[:50], x_recon[:50]]))
325     #img = combine_images(np.concatenate([x_recon[:50], x_recon[:50]]))
326     image = img * 255
327     Image.fromarray(image.astype(np.uint8)).save(args.save_dir + "/
            real_and_recon.png")
328     print()
329     print('Reconstructed images are saved to %s/real_and_recon.png' % args.
            save_dir)
330     print('-' * 30 + 'End: test' + '-' * 30)
331     plt.clf()
332     plt.imshow(plt.imread(args.save_dir + "/real_and_recon.png"))
333     plt.show()
334
335
336 if __name__ == "__main__":
337
338     # setting the hyper parameters
339     parser = argparse.ArgumentParser(description="Capsule Network on COVID
            dataset.")
340     parser.add_argument('--epochs', default=30, type=int)
341     parser.add_argument('--batch_size', default=32, type=int)
342     parser.add_argument('--img_width', default=256, type=int)
343     parser.add_argument('--img_height', default=256, type=int)
344     parser.add_argument('--lr', default=0.001, type=float,
345                         help="Initial learning rate")
346     parser.add_argument('--lr_decay', default=0.9, type=float,
347                         help="The value multiplied by lr at each epoch. Set
                                a larger value for larger epochs")
348     parser.add_argument('--lam_recon', default=32.768, type=float,  #gaby 5
            32.768,
349                         help="The coefficient for the loss of decoder")
350     parser.add_argument('-r', '--routings', default=3, type=int,
351                         help="Number of iterations used in routing
```

```
                            algorithm. should > 0")
352     parser.add_argument('--shift_fraction', default=0.1, type=float,
353                         help="Fraction of pixels to shift at most in each
                                direction.")
354     parser.add_argument('--debug', action='store_true',
355                         help="Save weights by TensorBoard")
356     parser.add_argument('--save_dir', default='./gaby3')
357     parser.add_argument('-t', '--testing', action='store_true',
358                         help="Test the trained model on testing dataset")
359     parser.add_argument('--digit', default=5, type=int,
360                         help="Digit to manipulate")
361     parser.add_argument('-w', '--weights', default=None,
362                         help="The path of the saved weights. Should be
                                specified when testing")
363     args = parser.parse_args()
364     print(args)
365
366     if not os.path.exists(args.save_dir):
367         os.makedirs(args.save_dir)
368
369     ######  ESTE ES EL PROGRAMA PRINCIPAL  ######
370     # load data
371     train_ds, val_ds =get__training_dataset()
372
373     print("Experimento")
374     x, y = next(iter(train_ds))
375     print(x[0].shape[1:])   #con este uno le quito el primer elemento al
            vector
376
377     # define model
378     model, eval_model = CapsNet(input_shape=x[0].shape[1:],  #channels,
            grayscale or rgb
379                                                 n_class=3,  #3 y_train[1]
380                                                 routings=args.routings,
                                                        #3
381                                                 batch_size=args.
```

```
                                                    batch_size)   #32
382
383     model.summary()
384
385     if args.weights is not None:  # init the model weights with provided
            one
386         model.load_weights(args.weights)
387     if not args.testing:
388         print("COMENTARIO!!!")
389         train(model=model, data=(train_ds, val_ds), args=args)
390     else:  # as long as weights are given, will run testing
391         if args.weights is None:
392             print('No weights are provided. Will test using random
                    initialized weights.')
393         print("HOLA!!!")
394         #test_ds = get_testing_dataset()
395         test(model=eval_model, args=args)
```

A.3 CapsNet Functions

```
1
2 #importo las librerias necesarias
3 import tensorflow as tf
4 import os
5 import tensorflow.keras.backend as K
6 from tensorflow.keras import layers,initializers
7
8
9 def squash(vectors, axis=-1):
10  """
11      The non-linear activation used in Capsule. It drives the length of a
            large vector to near 1 and small vector to 0
12      :param vectors: some vectors to be squashed, N-dim tensor
13      :param axis: the axis to squash
14      :return: a Tensor with same shape as input vectors
```

```
"""
 s_squared_norm = tf.math.reduce_sum(tf.square(vectors), axis, keepdims=
     True)
 scale = s_squared_norm / (1 + s_squared_norm) / tf.sqrt(s_squared_norm + K
     .epsilon())
 return scale * vectors


#bloque que declara Primary Caps

def PrimaryCaps(inputs, dim_capsule, n_channels, kernel_size, strides,
    padding):
    """
    Aplica una convolucion de la profundidad del numero de canales y
        concatenamos todas las capsulas
    inputs es un 4D tensor, shape=[None, width, height, channels]
    dim_capsules: la dimension del vector de salida de las capsulas 8
    n_channels: el numero de tipos de capsula  32
    return: output tensor, shape = [None, num_capsule, dim_capsule] o [None
        , 1152,8]
    """
    output = tf.keras.layers.Conv2D(dim_capsule*n_channels, kernel_size,
        strides=strides, padding=padding,name='primarycap' )(inputs)
    #filters=dim_capsule*n_channels(8*32=256), la salida es (none, 6,6,256)
    outputs = tf.keras.layers.Reshape((-1, dim_capsule), name='
        primarycaps_reshape')(output)
    #puede haber error en elprimer parametro, puede ser [] en lugar de ()
    return tf.keras.layers.Lambda(squash, name='primarycap_squash')(outputs
        )


#bloque Capsule Layer, donde se realiza el dynamic routing

class CapsuleLayer(tf.keras.layers.Layer):
    """
    The capsule layer. It is similar to Dense layer. Dense layer has `
```

```
        in_num` inputs, each is a scalar, the output of the
43      neuron from the former layer, and it has `out_num` output neurons.
        CapsuleLayer just expand the output of the neuron
44      from scalar to vector. So its input shape = [None, input_num_capsule,
        input_dim_capsule] and output shape = \
45      [None, num_capsule, dim_capsule]. For Dense Layer, input_dim_capsule =
        dim_capsule = 1.
46      :param num_capsule: number of capsules in this layer
47      :param dim_capsule: dimension of the output vectors of the capsules in
        this layer
48      :param routings: number of iterations for the routing algorithm
49      """
50
51      def __init__(self, num_capsule, dim_capsule, routings=3,
        kernel_initializer='glorot_uniform',**kwargs):
52          super(CapsuleLayer, self).__init__(**kwargs)
53          self.num_capsule = num_capsule  #3 clases de covid
54          self.dim_capsule = dim_capsule  #16
55          self.routings = routings
56          self.kernel_initializer = tf.keras.initializers.get(
                kernel_initializer)  #puede haber error
57
58      def build(self, input_shape):
59          assert len(input_shape) >= 3, "The input Tensor should have shape=[
                None, input_num_capsule, input_dim_capsule]"
60          self.input_num_capsule = input_shape[1]
61          self.input_dim_capsule = input_shape[2]
62
63          # Transform matrix, from each input capsule to each output capsule,
                there's a unique weight as in Dense layer.
64          self.W = self.add_weight(shape=[self.num_capsule, self.
                input_num_capsule,
65                                          self.dim_capsule, self.
                                                input_dim_capsule],
66                                   initializer=self.kernel_initializer,
67                                   name='W')
```

```
68
69          self.built = True
70
71      def call(self, inputs, training=None):
72          # inputs.shape=[None, input_num_capsule, input_dim_capsule]
73          # inputs_expand.shape=[None, 1, input_num_capsule,
                input_dim_capsule, 1]
74          inputs_expand = tf.expand_dims(tf.expand_dims(inputs, 1), -1)
75
76          # Replicate num_capsule dimension to prepare being multiplied by W
77          # inputs_tiled.shape=[None, num_capsule, input_num_capsule,
                input_dim_capsule, 1]
78          inputs_tiled = tf.tile(inputs_expand, [1, self.num_capsule, 1, 1,
                1])
79
80          # Compute `inputs * W` by scanning inputs_tiled on dimension 0.
81          # W.shape=[num_capsule, input_num_capsule, dim_capsule,
                input_dim_capsule]
82          # x.shape=[num_capsule, input_num_capsule, input_dim_capsule, 1]
83          # Regard the first two dimensions as `batch` dimension, then
84          # matmul(W, x): [..., dim_capsule, input_dim_capsule] x [...,
                input_dim_capsule, 1] -> [..., dim_capsule, 1].
85          # inputs_hat.shape = [None, num_capsule, input_num_capsule,
                dim_capsule]
86          inputs_hat = tf.squeeze(tf.map_fn(lambda x: tf.matmul(self.W, x),
                elems=inputs_tiled))
87
88           # Begin: Routing algorithm
                ---------------------------------------------------------------#
89          # The prior for coupling coefficient, initialized as zeros.
90          # b.shape = [None, self.num_capsule, 1, self.input_num_capsule].
91          b = tf.zeros(shape=[inputs.shape[0], self.num_capsule, 1, self.
                input_num_capsule])
92
93          assert self.routings > 0, 'The routings should be > 0.'
```

```
94          for i in range(self.routings):
95              # c.shape=[batch_size, num_capsule, 1, input_num_capsule]
96              c = tf.nn.softmax(b, axis=1)
97
98              # c.shape = [batch_size, num_capsule, 1, input_num_capsule]
99              # inputs_hat.shape=[None, num_capsule, input_num_capsule,
                    dim_capsule]
100             # The first two dimensions as 'batch' dimension,
101             # then matmal: [..., 1, input_num_capsule] x [...,
                    input_num_capsule, dim_capsule] -> [..., 1, dim_capsule].
102             # outputs.shape=[None, num_capsule, 1, dim_capsule]
103             outputs = squash(tf.matmul(c, inputs_hat))  # [None, 10, 1, 16]
104
105             if i < self.routings - 1:
106                 # outputs.shape =  [None, num_capsule, 1, dim_capsule]
107                 # inputs_hat.shape=[None, num_capsule, input_num_capsule,
                        dim_capsule]
108                 # The first two dimensions as 'batch' dimension, then
109                 # matmal:[..., 1, dim_capsule] x [..., input_num_capsule,
                        dim_capsule]^T -> [..., 1, input_num_capsule].
110                 # b.shape=[batch_size, num_capsule, 1, input_num_capsule]
111                 b += tf.matmul(outputs, inputs_hat, transpose_b=True)
112         # End: Routing algorithm
                -----------------------------------------------------------------------#
113
114         return tf.squeeze(outputs)   #quita los vectores con dimension 1
115
116     def compute_output_shape(self, input_shape):
117         return tuple([None, self.num_capsule, self.dim_capsule])
118
119     def get_config(self):
120         config = {
121             'num_capsule': self.num_capsule,
122             'dim_capsule': self.dim_capsule,
123             'routings': self.routings
```

```
124         }
125         base_config = super(CapsuleLayer, self).get_config()
126         return dict(list(base_config.items()) + list(config.items()))
127 
128 
129 class Length(tf.keras.layers.Layer):
130     """
131     Compute the length of vectors. This is used to compute a Tensor that
            has the same shape with y_true in margin_loss.
132     Using this layer as model's output can directly predict labels by using
             'y_pred = np.argmax(model.predict(x), 1)'
133     inputs: shape=[None, num_vectors, dim_vector]
134     output: shape=[None, num_vectors]
135     """
136     def call(self, inputs, **kwargs):
137         return tf.sqrt(tf.reduce_sum(tf.square(inputs), -1) + K.epsilon())
138 
139     def compute_output_shape(self, input_shape):
140         return input_shape[:-1]
141 
142     def get_config(self):
143         config = super(Length, self).get_config()
144         return config
145 
146 class Mask(tf.keras.layers.Layer):
147     """
148     Mask a Tensor with shape=[None, num_capsule, dim_vector] either by the
            capsule with max length or by an additional
149     input mask. Except the max-length capsule (or specified capsule), all
            vectors are masked to zeros. Then flatten the
150     masked Tensor.
151     For example:
152         ```
153         x = keras.layers.Input(shape=[8, 3, 2])  # batch_size=8, each
                sample contains 3 capsules with dim_vector=2
154         y = keras.layers.Input(shape=[8, 3])  # True labels. 8 samples, 3
```

```
            classes, one-hot coding.
155         out = Mask()(x)  # out.shape=[8, 6]
156         # or
157         out2 = Mask()([x, y])  # out2.shape=[8,6]. Masked with true labels
                y. Of course y can also be manipulated.
158         ```
159     """
160     def call(self, inputs, **kwargs):
161         if type(inputs) is list:  # true label is provided with shape = [
                None, n_classes], i.e. one-hot code.
162             assert len(inputs) == 2
163             inputs, mask = inputs
164         else:  # if no true label, mask by the max length of capsules.
                Mainly used for prediction
165             # compute lengths of capsules
166             x = tf.sqrt(tf.reduce_sum(tf.square(inputs), -1))
167             # generate the mask which is a one-hot code.
168             # mask.shape=[None, n_classes]=[None, num_capsule]
169             mask = tf.one_hot(indices=tf.argmax(x, 1), depth=x.shape[1])
170
171         # inputs.shape=[None, num_capsule, dim_capsule]
172         # mask.shape=[None, num_capsule]
173         # masked.shape=[None, num_capsule * dim_capsule]
174         masked = K.batch_flatten(inputs * tf.expand_dims(mask, -1))
175         return masked
176
177     def compute_output_shape(self, input_shape):
178         if type(input_shape[0]) is tuple:  # true label provided
179             return tuple([None, input_shape[0][1] * input_shape[0][2]])
180         else:  # no true label provided
181             return tuple([None, input_shape[1] * input_shape[2]])
182
183     def get_config(self):
184         config = super(Mask, self).get_config()
185         return config
```

A.4 Utils code

```
import numpy as np
from matplotlib import pyplot as plt
import csv
import math
import pandas

def plot_log(filename, show=True):

    data = pandas.read_csv(filename)

    fig = plt.figure(figsize=(4,6))
    fig.subplots_adjust(top=0.95, bottom=0.05, right=0.95)
    fig.add_subplot(211)
    for key in data.keys():
        if key.find('loss') >= 0 and not key.find('val') >= 0:  # training
            loss
            plt.plot(data['epoch'].values, data[key].values, label=key)
    plt.legend()
    plt.title('Training loss')

    fig.add_subplot(212)
    for key in data.keys():
        if key.find('acc') >= 0:  # acc
            plt.plot(data['epoch'].values, data[key].values, label=key)
    plt.legend()
    plt.title('Training and validation accuracy')

    # fig.savefig('result/log.png')
    if show:
        plt.show()


def combine_images(generated_images, height=None, width=None):
```

```
33      num = generated_images.shape[0]
34      if width is None and height is None:
35          width = int(math.sqrt(num))
36          height = int(math.ceil(float(num)/width))
37      elif width is not None and height is None:  # height not given
38          height = int(math.ceil(float(num)/width))
39      elif height is not None and width is None:  # width not given
40          width = int(math.ceil(float(num)/height))
41
42      shape = generated_images.shape[1:3]
43      image = np.zeros((height*shape[0], width*shape[1]),
44                       dtype=generated_images.dtype)
45      for index, img in enumerate(generated_images):
46          i = int(index/width)
47          j = index % width
48          image[i*shape[0]:(i+1)*shape[0], j*shape[1]:(j+1)*shape[1]] = \
49              img[:, :, 0]
50      return image
51
52  if __name__=="__main__":
53      plot_log('result/log.csv')
```