UNIVERSITY OF THE AEGEAN

MASTER THESIS

Sparse Multi-label Classification of Medical Images Using Deep Convolutional Neural Networks

Author:
Euangelos LINARDOS

Supervisor:
Ergina KAVALLIERATOU

Submitted in part fulfilment of the requirements
for the Master of Science degree in
Intelligent Information Systems

Artificial Intelligence Laboratory
Department of Information and Communication Systems Engineering
School of Engineering
University of the Aegean

February 2, 2021

Declaration of Originality

I, Euangelos Linardos, declare that this master thesis entitled, "Sparse Multi-label Classification of Medical Images Using Deep Convolutional Neural Networks", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Master of Science degree at the University of the Aegean.
- Where any part of this master thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this master thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where this master thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"The most interesting applications of machine learning sometimes come from people with very little technical background. The intersection of tech and art is where our humanity can really shine."

François Chollet

UNIVERSITY OF THE AEGEAN

Abstract

School of Engineering

Department of Information and Communication Systems Engineering

Artificial Intelligence Laboratory

Master of Science

Sparse Multi-label Classification of Medical Images Using Deep Convolutional Neural Networks

by Euangelos LINARDOS

The recent advancements in imaging technologies have improved clinician's ability to detect, diagnose, and treat diseases. As an example, radiologists routinely interpret medical images and summarize their findings in the form of radiology reports. The mapping of visual information present in medical images to the condensed textual description is a tedious, time-consuming, expensive, and error-prone task. The development of methods that can automatically detect the presence and location of medical concepts in medical images can improve radiologists' efficiency, reduce the burden of manual interpretation, and help reduce diagnostic errors.

In this master thesis, we deal with the challenging task of medical image tagging (or labeling), which aims to identify medical terms (or concepts) in medical images, and with the ultimate goal of helping physicians to generate medical reports from medical images. In particular, we propose a variation of convolutional neural networks for sparse multi-label classification to predict relevant concepts present in images, and we test it against all recent datasets from the ImageCLEFmed Caption task (i.e., 2017, 2018, 2019, and 2020). The proposed system outperformed all winning teams in terms of F_1 score between system predicted and ground truth concepts. We present our work with data analysis, experimental results, comparisons between the different hyper-parameters and network architectures, and last but not least, with a short discussion on future steps.

Keywords: *Machine Learning, Deep Learning, Image Processing, Computer Vision, Medical Imaging, Convolutional Neural Networks, Multi Labels, Sparse Labels, ImageCLEFmed, Concept Detection, Keras, TensorFlow*.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ

Περίληψη

Πολυτεχνική Σχολή

Τμήμα Μηχανικών Πληροφοριακών και Επικοινωνιακών Συστημάτων

Εργαστήριο Τεχνητής Νοημοσύνης

Μεταπτυχιακό Δίπλωμα Ειδίκευσης

Αυτόματη Επισήμανση Ιατρικών Εικόνων Πολλαπλών Ετικετών
με Χρήση Συνελικτικών Νευρωνικών Δικτύων

από τον Ευάγγελο ΛΙΝΑΡΔΟ

Οι πρόσφατες εξελίξεις στις τεχνολογίες απεικόνισης έχουν βελτιώσει την ικανότητα των ιατρών να εντοπίζουν, διαγιγνώσκουν, και θεραπεύουν ασθένειες. Για παράδειγμα, οι ακτινολόγοι καλούνται καθημερινά να εξετάσουν ιατρικές εικόνες και να συνοψίσουν τα ευρήματά τους υπό την μορφή ακτινολογικών αναφορών. Η χαρτογράφηση της οπτικής πληροφορίας που εντοπίζεται στις ιατρικές εικόνες σε συμπυκνωμένες ιατρικές αναφορές είναι μια επίπονη, χρονοβόρα, δαπανηρή, και επιρρεπής σε λάθη διαδικασία. Η ανάπτυξη μεθόδων που μπορούν να ανιχνεύσουν αυτόματα την παρουσία και τη θέση διαφόρων ιατρικών εννοιών στις ιατρικές εικόνες μπορεί να βελτιώσει την αποτελεσματικότητα των ιατρών, να μειώσει το φορτίο της χειροκίνητης ερμηνείας, και να βοηθήσει στην μείωση των διαγνωστικών λαθών.

Στην παρούσα μεταπτυχιακή εργασία, ασχολούμαστε με το απαιτητικό πρόβλημα της σήμανσης ιατρικών εικόνων, η οποία στοχεύει στον εντοπισμό ιατρικών εννοιών σε ιατρικές εικόνες, και έχει ως απώτερο στόχο την υποστήριξη των ιατρών στην δημιουργία ιατρικών αναφορών από ιατρικές εικόνες. Πιο συγκεκριμένα, προτείνουμε μια παραλλαγή των συνελικτικών νευρωνικών δικτύων για ταξινόμηση πολλαπλών ετικετών αραιών διανυσμάτων με στόχο την πρόβλεψη συναφών εννοιών που εντοπίζονται σε εικόνες, και τη δοκιμάζουμε σε όλα τα πρόσφατα σύνολα δεδομένων του ImageCLEFmed Caption διαγωνισμού (δηλ., 2017, 2018, 2019, και 2020). Η προτεινόμενη μέθοδος ξεπέρασε τους νικητές όλων των παλαιότερων διαγωνισμών βάση της F_1 μετρικής, μεταξύ των προβλέψεων του συστήματος και των δεδομένων αλήθειας. Παρουσιάζουμε τη δουλειά μας με ανάλυση δεδομένων, πειραματικά αποτελέσματα, συγκρίσεις μεταξύ των διαφόρων υπερ-παραμέτρων και αρχιτεκτονικών δικτύων, καθώς τέλος, με μια σύντομη αναφορικά σε μελλοντικές κατευθύνσεις.

Λέξεις-κλειδιά: Μηχανική Μάθηση, Βαθιά Μάθηση, Επεξεργασία Εικόνας, Υπολογιστική Όραση, Ιατρική Απεικόνιση, Συνελικτικά Νευρωνικά Δίκτυα, Πολλαπλές Ετικέτες, Αραιές Ετικέτες, *ImageCLEFmed*, Ανίχνευση Έννοιας, *Keras*, *TensorFlow*.

Acknowledgements

First and foremost, I'd like to thank my advisor, Assoc. Prof. Ergina Kavallieratou, for her guidance throughout this master thesis at the University of the Aegean as well as for encouraging me to participate multiple times in ImageCLEFmed Caption task. Many thanks also go to the committee members, Prof. Efstathios Stamatatos and Asst. Prof. Andreas Papasalouros, for their helpful comments and ideas during my master thesis presentation. Furthermore, I'd like to thank the board of AWS Cloud Credits for Research for offering me more than enough credits to run my resource-intensive experiments on their powerful GPUs. Last but not least, I'm deeply grateful for the Keras and TensorFlow communities for providing such excellent tools and for supporting users in a friendly manner.

Contents

List of Figures

List of Tables

List of Gists

List of Abbreviations

AI	**A**rtificial **I**ntelligence
AMI	**A**mazon **M**achine **I**mage
ANN	**A**rtificial **N**eural **N**etwork
AWS	**A**mazon **W**eb **S**ervices
BLAS	**B**asic **L**inear **A**lgebra **S**ubprograms
BN	**B**atch **N**ormalization
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CS	**C**omputer **S**cience
CV	**C**omputer **V**ision
DL	**D**eep **L**earning
DSC	**D**epthwise **S**eparable **C**onvolutions
DT	**D**ecision **T**ree
EC2	**E**lastic **C**ompute **C**loud
FE	**F**eature **E**ngineering
FFNN	**F**eed **F**orward **N**eural **N**etwork
GD	**G**radient **D**escent
GP	**G**radient **P**ropagation
GPU	**G**raphics **P**rocessing **U**nit
HSV	**H**ue **S**aturation **V**alue
IDE	**I**ntegrated **D**evelopment **E**nvironment
ILSVRC	**I**mage**N**et **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
JPEG	**J**oint **P**hotographic **E**xpert **G**roup
JPG	**J**oint **P**hotographic **G**roup
ML	**M**achine **L**earning
MLP	**M**ulti**L**ayer **P**erceptron
MNIST	**M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology
NN	**N**eural **N**etwork
OS	**O**perating **S**ystem
PMCOAS	**P**ub**M**ed **C**entral **O**pen **A**ccess **S**ubset
RAM	**R**andom **A**ccess **M**emory
RC	**R**esidual **C**onnections
RGB	**R**ed **G**reen **B**lue
RNN	**R**ecurrent **N**eural **N**etwork
ROCO	**R**adiology **O**bjects in **CO**ntext
SL	**S**hallow **L**earning
SSH	**S**ecure **SH**ell
SSL	**S**ecure **S**ockets **L**ayer
SVM	**S**upport **V**ector **M**achine
UI	**U**ser **I**nterface
UMLS	**U**nified **M**edical **L**anguage **S**ystem
URL	**U**niform **R**esource **L**ocator

To my beloved wife Eleftheria and my wonderful son Anargyros for encouraging and supporting me throughout this long journey. Thank you for the love, energy, and dedication during long days, nights, and weekends of exploring this exciting new field. I really couldn't have done it without you.

Chapter 1

Introduction

In this thesis, we explore the *Concept Detection* task, which is part of the *ImageCLEFmed* campaign. The task was first introduced in 2017 and has remained under the same format ever since (Eickhoff et al., 2017; Herrera et al., 2018; Pelka et al., 2019). Note, however, this is not the case for the datasets as they differ significantly in each edition, nevertheless always a subset of the *ROCO* dataset (Pelka et al., 2018). The *ROCO* dataset contains radiology images originating from the *PMCOAS* (Roberts, 2001), where each image is associated with one or more *UMLS* labels.

From a technical standpoint, this thesis's topic lies within the field of *multi-label image classification*. There are two caveats, though: one is that the distribution of labels is highly *skewed*, and the other is that the representation vector of labels is extremely *sparse*. These two properties make this task a tough one to solve. Our study, which is heavily inspired by the winning teams' work in all previous editions (Katsios and Kavallieratou, 2017; Pinho and Costa, 2018; Kougia, Pavlopoulos, and Androutsopoulos, 2019), will focus on applying *transfer learning* through an exhaustive *grid search* from a list of pre-trained models and a range of hyper-parameter values.

1.1 Background

Two of the most common image classification tasks are *multi-class* image classification and *multi-label* image classification. Each image belongs to exactly *one class* in the former, whereas the latter is a more challenging task where each image may have *multiple labels* associated with it. The multi-label image classification is an interesting yet complex *CV* task where the goal is to infer whether the image consists of specific features. The *DL* model is trained to recognize these features in an image by feeding them in the form of text labels during the training phase (Chen et al., 2018a; Chen et al., 2018b; Durand, Mehrasa, and Mori, 2019; Deng et al., 2009; Everingham et al., 2010; Ge, Yang, and Yu, 2018).

Even though recent breakthroughs in deep *CNNs* have pushed the boundaries on multi-label image classification, it remains a difficult task to overcome due to each image having more than a single label of interest. The overlap of multiple labels makes it essential to model accurate label dependencies for multi-label image classification (Chen et al., 2018a; Chen et al., 2019; George and Floerkemeier, 2014; Gong et al., 2014). A common approach to this problem, which was also adopted in this thesis, is to model the multiple label classification problem into multiple *binary* classification problems by considering each label *independent* of each other.

1.2 Motivation

Physicians' daily routine typically involves the *examination* of a vast amount of medical images toward the *diagnosis* of potential diseases. This is a *time-consuming* and *error-prone* process; at other times, there may be not enough experienced physicians to deal with such demanding decision-making. In order to help the diagnostic process, *CV* techniques, combined with recent advances in *DL*, can significantly assist the interpretation of medical images. Automatic methods can significantly reduce medical *errors* and benefit the medical departments tremendously by reducing the exams' *cost* and accelerating the diagnoses' *speed* (Bates et al., 2001; Lee et al., 2017).

Among the tasks that can be applied to medical images (Litjens et al., 2017) to assist the diagnostic process is medical image *classification*. As the name reveals, this task is all about assigning medical *labels* to an *image* so that physicians can focus on the most relevant medical terms (Shin et al., 2016). Despite the task's importance for the common good, medical images are not hugely available or accessible yet; hence, community research is currently limited (Oakden-Rayner, 2019). However, there is a growing interest in the *automatic* analysis of medical images nowadays. In this thesis, we study *four* popular datasets for medical image classification by implementing our own solution and comparing the results with the *winning* team from the corresponding campaign.

To recap, *multi-modal* approaches have been shown to achieve better results in image classification (Pelka, Nensa, and Friedrich, 2018). As the interpretation of medical images' knowledge is a *time-consuming* and *error-prone* process, there is a considerable need for *automatic* methods that can approximate this *mapping* from an image to one or more labels. The more image characteristics are known, the more structured the radiology scans are; hence, the more *accurate* the radiologists are regarding interpretation.

1.3 Challenges

During the past few years, we have seen some significant steps forward within the image classification field. In 2012, the system that utilized a *DL* approach outperformed all other methods in *ILSVRC* and achieved a top-5 error rate of 15.4% in the *single-label* image classification task (Krizhevsky, Sutskever, and Hinton, 2012; Russakovsky et al., 2015; Schmidhuber, 2015). Four years later, in 2016, the winner of *ILSVRC* improved this result even further with a top-5 error rate of 2.9%, using a new design of *CNN*. The progress of solving *multi-label* image classification also moved forward, improving classification precision year after year (Wei et al., 2016; Ren et al., 2015). Note, however, that all these methods and architectures were designed and tested on standard image collections, like *ImageNet* (Russakovsky et al., 2015) or *PASCAL VOC* (Everingham et al., 2010), which were specifically created to compare proposed approaches and solutions in different papers (Wei et al., 2016; Oquab et al., 2014; Gong et al., 2014; Chatfield et al., 2014) - and that is exactly the problem.

Most of the datasets created by *labs* for research purposes differ from those owned by *companies* in various ways, such as the *distribution* of the labels, the *sparsity* of the representation vector, the number of systematic *errors*, and more. In other words, real-world companies are likely to have more *unique* and *unstructured* datasets. Therefore,

it's crucial to study how the state-of-the-art methods perform on such *exotic* datasets and, therefore, create a system that can *automatically* overcome these limitations.

1.4 Outline

The remainder of this thesis is organized as follows: **Chapter 2** presents the main concepts of *DL*. **Chapter 3** introduces the key ideas of *CV* from the *DL* perspective. **Chapter 4** analyzes the input data through exploratory analysis, demonstrates the evaluation framework and the methodology used, and, last but not least, presents the results of our experiments. Finally, **Chapter 5** summarizes our work and proposes directions for the next steps.

Chapter 2

Fundamental Concepts of Deep Learning

2.1 Brief Introduction to Deep Learning

In the past few years, *AI* has been a subject of intense media hype. *AI* comes up in countless articles, often outside of technology-minded publications. We are promised a future that sometimes painted in a **harsh light** and other times as **utopian**, where human jobs will be rare, and most economic activity will be handled by robots and agents. For a future or current practitioner of *AI*, it's crucial to be able to distinguish the **signal from the noise** so that we can tell breakthroughs from overhyped press releases. This chapter provides the essential context around *DL* and the ecosystem around it.

2.1.1 The Artificial Intelligence Landscape

First off, let's define what we mean when we talk about *AI* and how it relates to *ML* and *DL* (see figure 2.1).

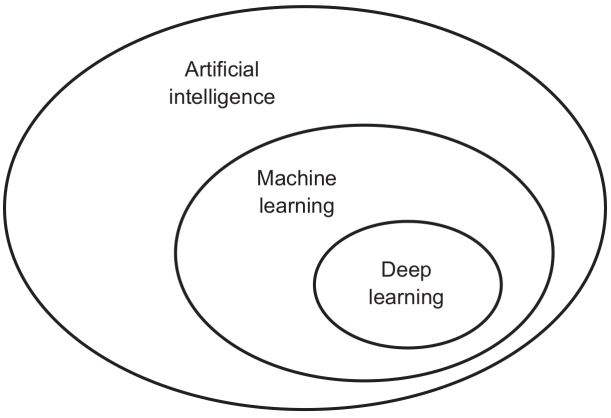

FIGURE 2.1: Artificial intelligence, machine learning, deep learning.

AI was born in the 50s when *CS* pioneers started asking whether computers could be made to **think**. A more concrete definition of the field would be the effort to **automate** intellectual tasks usually performed by humans. As such, *AI* can be considered a general field that encompasses not only *ML* and *DL*, but also many more approaches

that do not involve any learning. For a very long time, many experts believed that human-level *AI* could only be possible by a sufficiently large set of explicit rules. This approach is also known as **Symbolic AI** and reached its peak popularity during the expert systems boom of the 80s. Although this approach proved suitable for well-defined, logical problems, it turned out to be impractical for solving more complex, fuzzy problems, including but not limited to, image classification.

ML arises from the idea that a computer will automatically learn these rules by looking at the data rather than programmers crafting the data-processing rules by hand. As we already discussed, in classical programming, humans input the rules and the data to be processed according to these rules, and the answers are generated (see figure 2.1). However, in this new programming paradigm, humans input the data as well as the answers expected from the data, and the rules are generated. The new rules can then be applied to new data to produce original answers.

So technically, *ML* is all about **searching** for useful **representations** of some input data, within a predefined **space** of possibilities, using guidance from a **feedback signal**. It's a simple idea which, however, allows us to solve a remarkably broad range of intellectual tasks.

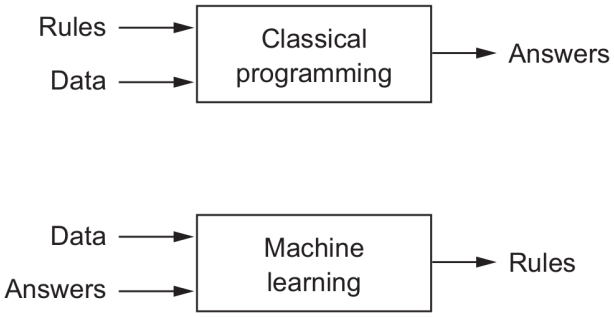

FIGURE 2.2: Clasical programming vs. machine learning.

In other words, a *ML* system is **trained** rather than explicitly **programmed**. It does so by reading many examples relevant to a task and trying to find a statistical structure that eventually allows the system to come up with rules for automating the particular problem.

Although *ML* only started to flourish in the 1990s, it quickly became the most popular and successful subfield of *AI*, a trend abetted by the availability of faster hardware and larger datasets. And unlike statistics, *ML* tends to deal with large, complex datasets for which classical statistical analysis would be impractical. As a result, *ML*, and especially *DL*, exhibit comparatively little mathematical theory, and hence, it's mostly engineering oriented.

DL is a specific subfield of *ML* and a new approach to learning representations from data with an emphasis on learning successive layers of increasingly meaningful representations. The **deep** in *DL* stands for this idea of consecutive layers of representations. Modern *DL* often involves tens or even hundreds of successive layers of representations, whereas classical *ML* approaches usually learn one or two layers of representations at most; hence, they are often called *SL*.

In most cases, these layered representations are learned via models called *NNs*, which are structured by stacking multiple layers on top of each other. Furthermore, although our understanding of the brain inspires some of the main concepts in *DL*, *DL* models

are not models of the brain; it's just a mathematical framework for learning representations from data.

Now let's examine how a network several layers deep (see figure 2.3) transforms an image of a number in order to recognize what number is shown in the image.

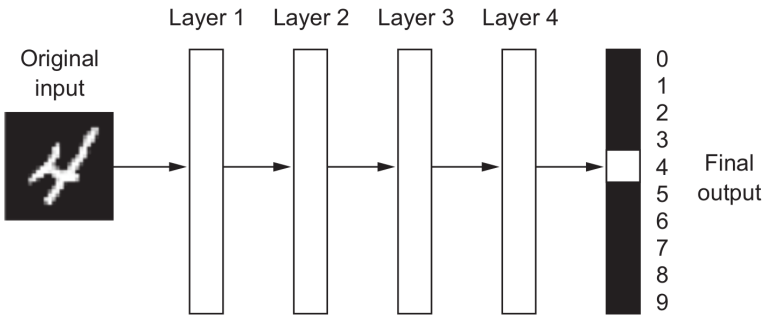

FIGURE 2.3: A deep neural network for digit classification.

As we can see in figure 2.4, a deep *NN* can be seen as a multistage **information distillation** operation, where information goes through serial filters and comes out increasingly refined.

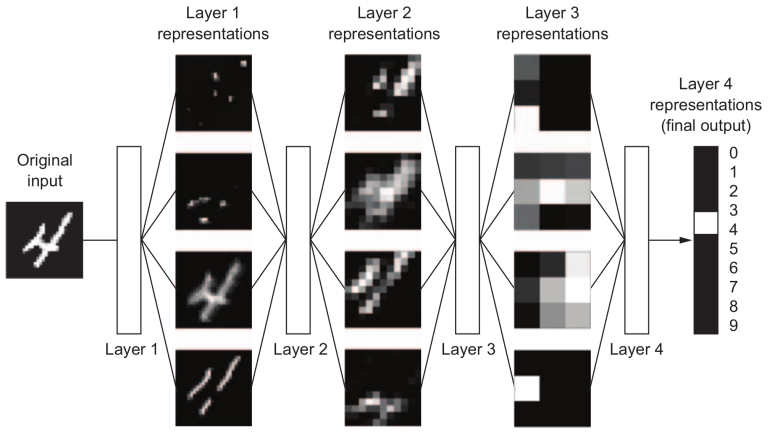

FIGURE 2.4: Deep representations learned by a classification model.

So technically, *DL* is a multistage way to learn data representations. It's a very simple idea, which, however, can end up look like magic!

2.1.2 Learning Deep Representations from Data

We already mentioned that **learning**, in the context of *ML*, describes an automated **search** process for better representations. So, basically, in order to perform *ML*, we need three main things:

- **Input data**: e.g., images such as *JPEG* or *JPG*.
- **Expected output**: e.g., tags such as *dog* or *cat*.
- **Score function**: e.g., measures such as F_1 or *precision* or *recall*.

The first two bullets should look familiar by now, so let's talk briefly about the third one. A **score function** is a way to estimate whether an algorithm is doing a good job or not. This task is necessary in order to determine the **distance** between the current output of the algorithm and the expected one. It's used as a **feedback signal** to adjust the way the algorithm works. This adjustment is what we call **learning**.

Recall that the primary goal of *ML* models is to transform input data into meaningful outputs, a process that is learned from exposure to known input and output examples. Therefore, the central problem in *ML* and *DL* is to transform data meaningfully, or in other words, to learn useful representations. But, what exactly do we mean when we say **representation**?

At its core, a **representation** is a different way to look at the data. For example, a color image can be encoded in various formats, including but not limited to *RGB* and *HSV*. Therefore, *ML* models are all about finding appropriate representations for the input data or putting it differently, transformations that make input data more amenable to the underlying task. Keep in mind, though, that *ML* algorithms are not always creative in finding these transformations; they are solely **searching** through a predefined set of operations called **hypothesis space**.

2.1.3 How Deep Learning Works in Three Figures

It should be evident by now that *ML* is all about *mapping* inputs to targets, which is done by processing many input-to-target examples, as well as that the mapping is done via a deep sequence of simple *layers* which are learned by exposure to those examples.

The instructions of what a layer should do to its input data are embedded in the layer's weights (see figure 2.5). Within this context, **learning** is the process of finding a set of weights for all layers in the network, such that the network will map inputs to targets as precisely as possible. The only problem is that a network can contain tens of millions of parameters. Therefore, finding the right value for all of them is a time-consuming and computationally-expensive task, as modifying one parameter will affect the behavior of all the others.

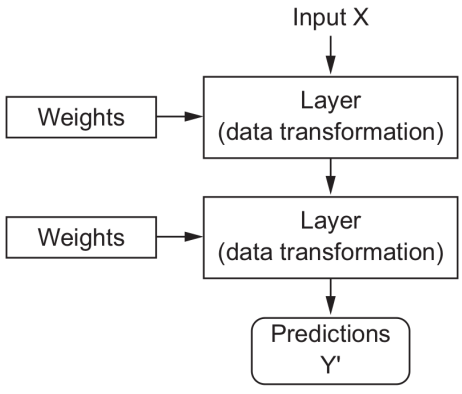

FIGURE 2.5: A neural network is parameterized by its weights.

In order to adjust the output of a *NN*, we must be able to estimate how far the predicted output is from the expected one. This is the job of the **loss function**, which takes the

predictions of the network along with the corresponding *targets*, and computes their in-between *distance* (see figure 2.6).

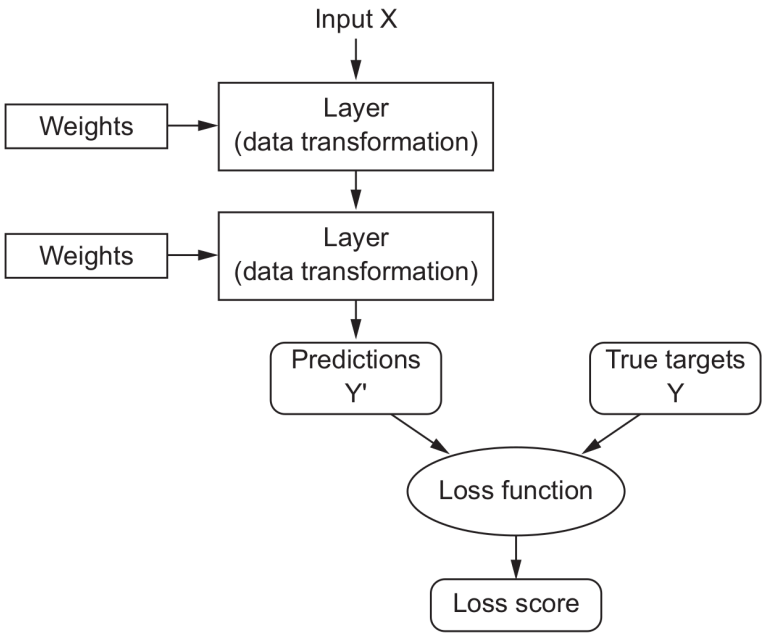

FIGURE 2.6: A loss function measures the quality of the network.

The underlying idea in *DL* is to use the *distance score* as a **feedback signal** to slightly control the weights, toward a direction that will minimize the score of the current example set (see figure 2.7). This is the job of the **optimizer**, which is based on the foundation of the **backpropagation** algorithm.

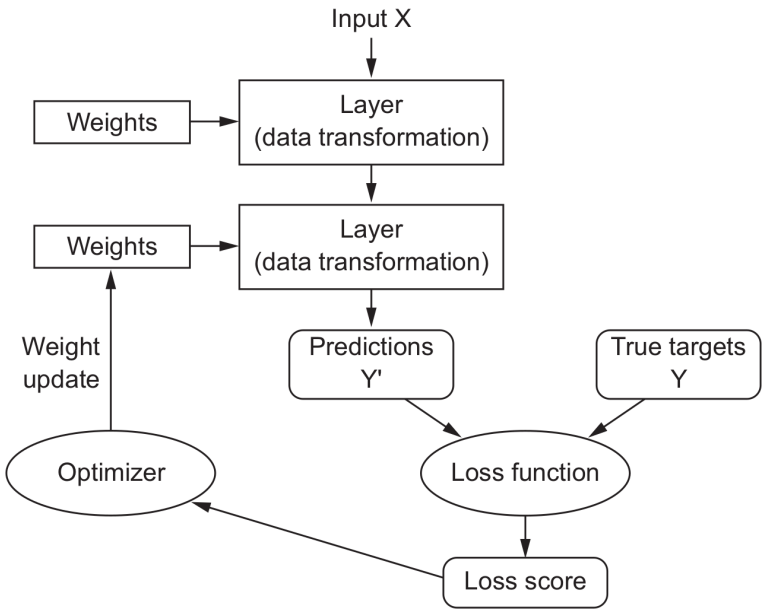

FIGURE 2.7: A loss score serves as feedback signal to adjust the weights.

At the beginning of the **learning** process, the weights of the network consist of random values. Because of that, the network only performs a series of arbitrary transformations, which in turn lead to a very high *loss score*. However, while the network

processes more-and-more examples, the weights are getting adjusted steadily toward an ever-decreasing *loss score*. This process is called **training loop**, which, repeated an adequate number of times, yields weights that minimize the *loss score*. As we have previously mentioned, *DL* is a simple mechanism that, once scaled, ends up looking like magic.

2.1.4 From Neural Networks to Deep Learning

DL has attracted a significant level of public attention and industry investment, unlike no other subfield in the long history of *AI*. Still, it's not the first successful form of *ML*. In fact, most *ML* algorithms used in the industry today are not *DL* algorithms. *DL* is not always the right tool for the job. For example, sometimes there is not enough data to train a model; other times the task is better solved by a different algorithm.

Although modern variants of *NNs* have completely substituted early efforts, it's useful to know how *DL* originated. The core ideas of *NNs* were studied as early as the 1950s; however, it took decades to get traction. For a long time, the missing part was an efficient way to train large *NNs*. The tipping point came in the mid-1980s when multiple researchers independently discovered the **backpropagation** algorithm and started applying it to *NNs*.

The first successful real-world application of *NNs* came in the late 1980s when the earlier ideas of *CNNs* combined with the **backpropagation** algorithm and applied to the problem of handwritten digits classification.

Around 2010, although the scientific community at large almost completely avoided *NNs*, several people still working on them started to make significant breakthroughs.

In 2011, researchers started to win academic image classification competitions with deep *NNs*, trained by *GPUs*. But the tipping point came in 2012 as part of the *ImageNet* large-scale image classification challenge. This challenge, which was especially tricky at the time, is made up of 1.4 million high-resolution color images and 1 thousand unique categories. In 2011, the top-5 accuracy of the winning model, which was solely based on classical *CV* approaches, was only 74.3%. Then, in 2012, a team of researchers achieved a top-5 accuracy of 83.6%. The competition has been dominated by deep *CNNs* every year since. By 2015, the winning team reached a top-5 accuracy of 96.4%, and the challenge was considered completely solved.

Since 2012, deep *CNNs* have become the go-to approach for all *CV* tasks. At major *CV* conferences in 2015 and 2016, it was nearly impossible to find presentations that did not involve *CNNs* in some form. *CNNs* have completely replaced *SVMs* and *DTs* in a wide range of applications ever since.

The main reason why *DL* took off so fast was that it offers better performance on a wide range of problems. Additionally, it solves problems much easier since it completely automates *FE*, probably the most critical and demanding step in *ML* workflows.

2.1.5 The Driving Forces Behind Deep Learning

Both the *CNNs* and the **backpropagation** algorithm were already well understood by the year 1989. Still, it took more than two decades for *DL* to take off. In general, *ML* advances are driven by three major forces:

- Faster hardware.
- Bigger datasets.
- Smarter algorithms.

Unlike **Mathematics** and **Physics**, where prominent breakthroughs can be made with pen and paper, *ML* is a purely **Engineering** science. That means, *ML* is guided by *experiments* rather than *theory*, and hence, algorithmic breakthroughs can only become possible when bigger datasets and faster hardware are available to try out new ideas.

During the 1990s and 2000s, commercial *CPUs* became three to four orders of magnitude faster. As a result, it's now possible to run small *DL* models on a laptop, something that would have been impossible 30 years ago.

In reality, though, common *DL* models require orders of magnitude more computational power than what a laptop has to offer. Luckily for us, during the 2000s, companies invested billions of dollars in developing extremely-fast and massively-parallel devices for the video game industry. These efforts came to benefit the scientific community too, when, in 2007, *NVIDIA* launched the *CUDA* programming interface. As a result, a small number of *GPUs* started replacing massive clusters of *CPUs* in a variety of high-performance applications.

Meanwhile, large tech companies started training *DL* models on clusters of hundreds of *GPUs*. The raw power of such clusters would never be possible without modern *GPUs*.

As far as data, in addition to the exponential growth in storage hardware technology over the past two decades, the game-changer has been the rise of the **Internet**, making it feasible to collect and distribute very big datasets for *ML*. Today, large tech companies work with image datasets that could not have been collected without the **Internet**. For example, user-generated image labels on *Flickr* have been a wealth of data for *CV* practitioners.

If there is one dataset that has been the driver for the acceleration of *DL* in *CV*, this is, of course, the *ImageNet* image database, which consists of 1.4 million public images along with 1 thousand unique categories.

Until the late 2000s, and in addition to the hardware and data challenges, we were lucking an efficient way to train very deep *NNs*. Therefore, *NNs* had to use one or two layers of representations at most. The main difficulty was that of *GP* through deep stacks of layers, and more specifically, the fact that the **feedback signal** that was used to train *NNs* would evaporate as the number of layers increased.

Luckily, the situation changed around 2010 with the arrival of various simple yet effective algorithmic improvements that allowed for more advanced *GP*:

- Activation functions.
- Weight-initialization schemes.
- Optimization schemes.

Only when these algorithmic improvements began to support training models with ten or more layers did *DL* start to shine.

Eventually, around 2014, even more sophisticated methods to help *GP* were discovered, including but not limited to, *BN*, *RC*, and *DSC*. Nowadays, we can train models that are thousands of layers deep from scratch in a matter of hours.

2.2 Neural Networks Building Blocks

In order for us to better understand *DL*, we first need to get familiar with a few simple yet powerful mathematical concepts, including but not limited to, *tensors*, *tensor operations*, *differentiation*, and *gradient descent*. As a result, the goal of this section is to build a good intuition around them without getting too much into details.

2.2.1 Quick Glance at a Neural Network

The fundamental building block of *NNs* is the **layer**, a data processing component that we can think of as a **filter** in *CV* terms; data goes in, and it comes out in a more usable form. More precisely, we can see the **layer** as a tool to extract representations from the input data. Moving forward, *DL* models primarily consist of chaining together simple layers that, when considered all together, implement a form of progressive data distillation.

To make a *DL* model ready for use, and in addition to layers, we also have to choose three important components, during compilation step:

- **Loss**: a function to *measure* the network *performance* on the training data so it can *adjust* itself in the right direction.
- **Optimizer**: a mechanism via which the network will *update* itself based on the *input data* and *loss function*.
- **Metric**: a function to *monitor* the network *performance* during training and testing phases for *informational* purposes only.

The precise role and specs for each of these concepts will be discussed in detail in the following sections.

2.2.2 Data Representations for Neural Networks

Most *ML* systems nowadays use *tensors* as their primitive data structure. In fact, *tensors* are fundamental to the field.

In essence, *tensors* are holders for numbers. We are all familiar with *matrices*, which are *2D tensors*. Think of *tensors* as a generalization of *matrices* to an arbitrary number of *dimensions*, also known as *axes*.

A tensor is characterized by three main features:

- **Axes**: e.g., a 3D tensor has three axes and a 2D matrix has two axes.
- **Shape**: e.g., a (3, 3, 5) 3D tensor, a (3, 5) matrix, a (5,) vector, or a () scalar.

- **Type**: e.g., float32, uint8, or float64.

Generally, color images have three dimensions, namely, *height*, *width*, and *depth*. On the contrary, grayscale images have a single color and hence depth. They can, therefore, be saved in *2D* matrixes, although, by convention, they are saved in *3D* tensors of an one-dimensional channel.

As we have already discussed, in *DL*, images are processed in batches. So, in practice, images are stored in *4D* tensors of shape *(samples, height, width, depth)*. For example, a batch of 32 grayscale images of size 128×128 are stored in a tensor of shape *(32, 128, 128, 1)*, whereas a batch of 32 color images of the same size are stored in a tensor of shape *(32, 128, 128, 3)* (see figure 2.8).

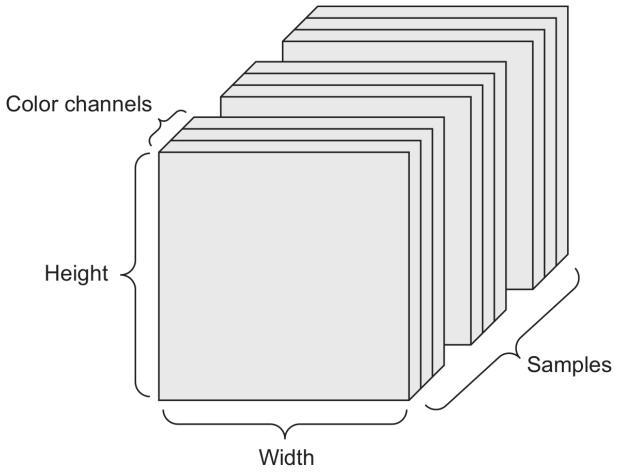

FIGURE 2.8: A 4D channels-first image data tensor.

Finally, it's worth mentioning that, in general, there are two conventions of shapes for image tensors; the *channels-first* and the *channels-last*. TensorFlow uses the latter convention, based on which the depth axis is placed at the end, i.e., *(samples, height, width, depth)*.

2.2.3 Anatomy of a Neural Network

In this section, we will take a closer look at the core components of *NNs* that we introduced earlier in this chapter:

- Layers and model.
- Input data, targets, and predictions.
- Loss function.
- Optimizer.

The visual interaction of these components is illustrated in figure 2.7. First of all, the *network* consists of *layers* that are connected with each other, and its purpose is to produce *predictions* from the *input data*. Then, the *loss function* compares the *predictions* to the *targets*, calculating a *loss value*, which essentially is a measure of how well the *predictions* match with the *targets*. Finally, the *optimizer* uses the *loss value* in order to adjust the *weights* of the *network* accordingly.

Since we now have a high-level idea of the purpose of each component, let's dive into the specifics.

The underlying component in *NNs* is the *layer*; a data-processing module that takes one or more tensors as input and returns one or more tensors as output. It's worthwhile to mention that some layers are *stateless*, but most often than not, they have a *state*. Moreover, some layers are more appropriate for certain tensor formats and data types than others. For instance, *image data*, which are stored in *4D* tensors, is usually processed by *2D* convolution layers.

As we already mentioned in a previous section, a *DL* model is a directed acyclic graph (a.k.a. *DAG*) of layers, with the most common example being a linear stack of layers, mapping a single input to a single output.

The architecture of a model defines a **hypothesis space**, and thus, by choosing one over the others, we restrict our **hypothesis space** to a certain series of *tensor operations*. Once there, the final step is to search for a good set of weights for all the tensors involved in these operations. Obviously, picking the right architecture is more an **art** than a **science**, and therefore, only practice can make us better *NN* practitioners.

Now, let's talk briefly about **activation functions** and why they are considered so important. But before we do so, let's examine a very simple network, like the one illustrated in figure 2.9.

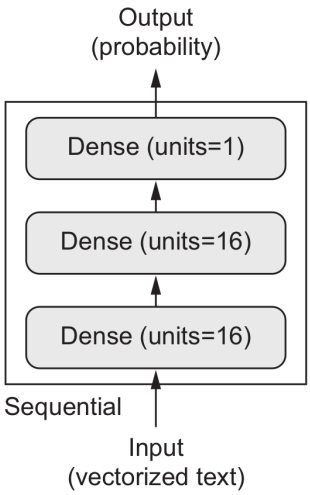

FIGURE 2.9: Sample three-layer network.

By the way, the network in figure 2.9 can be easily implemented in *Keras* as described in gist 2.1.

```
from keras import models, layers
base = models.Sequential()
base.add(layers.Dense(32, activation='relu', input_shape=(1000,)))
base.add(layers.Dense(32, activation='relu'))
base.add(layers.Dense(1, activation='sigmoid'))
```

GIST 2.1: Sample model definition in Keras.

Without an *activation function*, the *dense layer* would consist of two *linear operations* - i.e., a dot product and an addition - and would be expressed by $y = f(w \cdot x + b)$.

In such a scenario, the *hypothesis space* would be too confined and would not take advantage of the multiple representation layers because a deep stack of linear layers would still perform a linear operation. In order to get access to a wealthier *hypothesis space* that would benefit from deep representations, we need an *activation function*, the purpose of which is to introduce *non-linearity*. We are not done yet, though!

Firstly, it's extremely important to choose the right *loss function* for the right problem. For example, *multi-class @ single-label* classification problems work better with *categorical_crossentropy* loss, whereas *multi-class @ multi-label* classification problems work better with *binary_crossentropy* loss. So, in general, the rule of thumbs work pretty well for simple problems. On the contrary, though, truly new research problems often require truly new objective functions.

Secondly, it's important to choose a proper *optimizer*, the goal of which is to find the optimal *weights* that minimize the total *error*. In other words, it's an important mechanism toward the optimization of both weights and biases, and thus the minimization of the *loss value*. Obviously, there are several optimizers to choose from depending on the problem we are trying to solve, but almost always of some *GD* variant.

2.3 Machine Learning Best Practices

In this section, we will establish a solid framework for tackling *ML* problems. More specifically, we will combine various techniques, including *model evaluation*, *data preprocessing*, *feature engineering*, *class-imbalance handling*, into a comprehensive workflow for solving any supervised learning task using *NNs*.

2.3.1 Supervised Learning

The goal in *supervised learning* is to learn the *relationship* between the **inputs** and the **targets**. The most popular category of *supervised learning* is *classification*, which breaks down further into **binary**, **multiclass**, and **multilabell**. Almost all *ML* applications that are in the spotlight today - such as image classification - fell into this category.

It's, for sure, a complex domain that involves many specialized terms. We already came across some of them in the previous sections, and we will see more of them in a bit. They come with precise definitions with which we should all be familiar. So, let's get started.

- **Sample**: What goes into our model.
- **Prediction**: What comes out of our model.
- **Target**: What we expect our model to predict.
- **Error**: The distance between prediction and target.
- **Classes**: A set of possible labels to choose from.
- **Label**: A specific instance of a class annotation.
- **Ground**: All targets for a dataset.
- **Binary**: Each sample is categorized into two exclusive categories.

- **Multiclass**: Each sample is categorized into more than two exclusive categories.
- **Multilabel**: Each sample can be assigned multiple labels.
- **Batch**: A set of samples processed on training toward a gradient descent update.

2.3.2 Model Evaluation

We already know that the main challenge in *ML* projects is to build models that *generalize* well. Consequently, one of the first steps in *ML* workflows is to split the raw dataset into *train*, *validation*, and *test*. Skipping to do so, but rather evaluating the model on the same data it was trained, and a few epochs later, the model will begin to *overfit*: i.e., the performance of the model on unseen data will start degrading compared to the known one.

Having said that, let's discuss some useful techniques to avoid *overfitting*. The process is pretty straightforward. First, we train our model on the train data. Next, we evaluate it on the validation data. Note that we usually repeat the two steps again and again with different hyper-parameters. Finally, once our model is ready to go live, we evaluate it for one last time, but this time on the test data.

Someone may ask why wasting a portion of the data for validation when we can just use the test set? We will not get into details here, but the main reason is related to the *information leak* concept.

At the end of the day, we will end up with a model that performs artificially well on the validation data. However, what we should really care about is its performance on the unseen data. Therefore, the importance of the test set is vital. There are several techniques to split the raw data into three parts, so let's quickly study the most popular ones.

In the *simple hold-out validation* technique, we keep some fraction of the raw data separated as our test set. Then, we train on the first part and evaluate on the second. Schematically, this technique looks like figure 2.10.

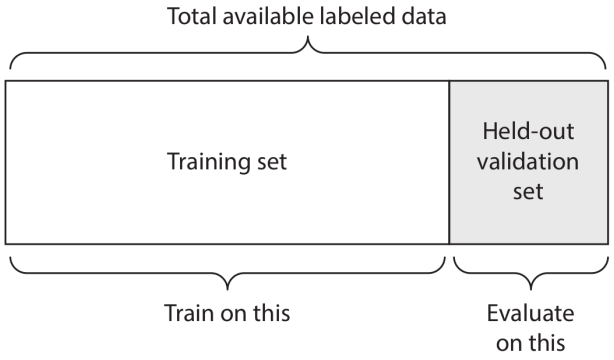

FIGURE 2.10: Simple hold-out validation.

This technique is fast to implement and execute, but it comes with a big flaw: if very little data is available, then both validation and test sets may contain too few samples, and, therefore, neither of them will be representative of the original data.

Fortunately, the *k-fold cross-validation* technique comes to the rescue, as it can partially address this flaw. On the negative side, it's more complex to implement, and it takes

significantly more time to execute. The main idea is to split the original dataset into K partitions of equal size. For each partition p, we train a model on the remaining K-1 partitions and evaluate it on partition p. The final score is then the average of all K scores obtained. Schematically, the *3-fold cross-validation* technique looks like figure 2.11.

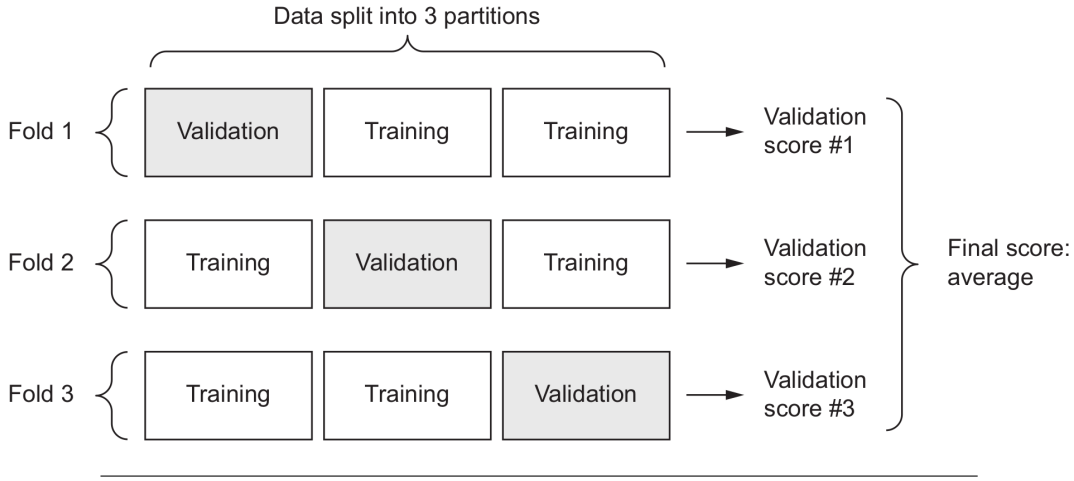

FIGURE 2.11: Three-fold cross-validation.

It's worth mentioning that a quick and dirty way to identify whether we should choose *k-fold cross-validation* over *simple hold-out validation* is based on the size of variation among the successive *shuffle-split-train-evaluate* rounds.

Before we finish off this section, it's important to keep an eye out for the following two rules when choosing an evaluation technique:

- **Data representativeness**: Both train and test sets should represent the raw data as good as possible. To ensure this, it's always suggested to randomly shuffle the raw data before splitting.
- **Data redundancy**: If some samples appear twice in the raw data, then we might end-up testing part of the train data. To avoid this, we should always make sure that train and validation sets are disjoint.

2.3.3 Data Pre-Processing

Before we deep dive into model development, we must tackle the critical question of how to prepare the data before feeding them into a *NN*. Most of these techniques are domain-specific, and thus, they will not be covered here; instead, we will only review the basics that are common to all domains.

As the name reveals, data pre-processing aims at making the data ready for consumption by *NNs*. This process involves multiple steps, including but not limited to, *vectorization*, *normalization*, and *handling missing values*. Let's look at each of them briefly.

Vectorization is the process where input data is turned into *tensors*. Recall that all inputs and targets in a *NN* must be tensors of *integer* or *floating-point* numbers.

Normalization is the process where input data become *small* and *homogenous*. Skipping this step will result in large gradient updates that will prevent the *NN* from converging. Especially in the context of *CV*, the general rule of thumb is:

- Normalize each feature independently to have a mean of 0.
- Normalize each feature independently to have a standard deviation of 1.

Handling Missing Values is the process where missing values are set to *0*, provided that *0* is not a meaningful value for the given problem. As a result, the *NN* will learn that value *0* means missing and thus, will start *ignoring* it.

2.3.4 Feature Engineering

In a nutshell, *FE* is all about making a problem *easier* to solve by describing it in a much *simpler* fashion. In order to achieve this, it's often required that we have a good understanding of the problem at hand.

It's a process that not only requires good knowledge of various *statisticcal methods* but most importantly, deep knowledge of the *underlying data*. This will allow the algorithm to work better by applying hardcoded transformations to the data before it goes into the model. In general, it's not reasonable to expect a model to learn from entirely arbitrary data. Instead, the data needs to be presented to the model in a way that will make training fast and efficient.

Before the *DL* era, *FE* used to be extremely critical because classical *SL* algorithms do not have *hypothesis spaces* rich enough to learn useful features by themselves. Thus, the way we fed the data to the algorithm was crucial to its success.

Nowdays, *DL* removes the need for most *FE* tasks because *NNs* are capable of extracting useful *patterns* from the data *automatically*. However, using *NNs* by no means implies that we do not have to worry about *FE*. In fact, it is and will always be important and relevant because good features always help us solve problems more elegant and with fewer:

- Computational resources.
- Training samples.

2.3.5 Class-Imbalance Handling

Not all datasets are perfect. In fact, we will be extremely lucky if we ever get a perfectly-balanced, real-world dataset. Most of the time, our data will have some level of class-imbalance, which is when each of our classes has a different number of examples.

Nevertheless, always remember that class-imbalance techniques are only important when we care about the *minority* classes. For instance, let's assume that we are working on a *multi-class* or a *multi-label* image classification task where class distribution looks like figure 2.12.

Initially, it may seem reasonable to balance our data, but soon enough, we may realize that we are not interested in the *minority* classes but rather in getting the highest possible *accuracy*. That means balancing our data will have a negative effect since most of the *accuracy* comes from classes with many examples. In other words, *minority* classes do not contribute much to our goal, and thus, balancing is not required and must be avoided.

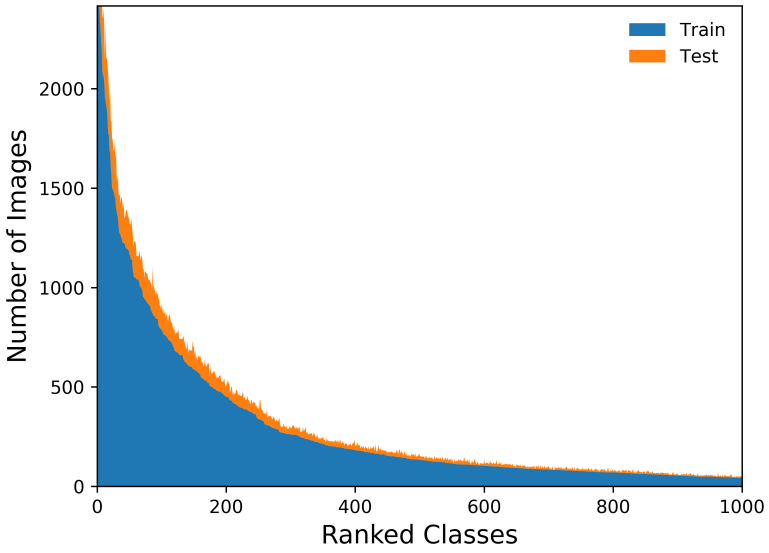

FIGURE 2.12: Class distributions for train and test.

Now let's assume we work on a task where balancing our data is critical. In such a scenario, the following three techniques will come in handy.

The **weight balancing** method balances the data by adjusting the weight of each class when computing the loss. Typically, each class in *loss function* will carry equal weight. However, sometimes we might want certain classes to hold more weight if they are more important to us. This can be done by *multiplying* the loss of each class by a class-specific *scalar*.

The main idea behind the **focal loss** method is that instead of giving equal weighting to all training classes, it *down-weights* the *well-classified* ones. This results in putting more emphasis on the classes that are hard to classify. In other words, given any imbalance dataset, the *majority* classes will quickly become *well-classified* since we have much more data for them. The **focal loss** comes into play when we also want to achieve high accuracy in the *minority* classes. It does so by giving them more relative weight during training. Check out figure 2.13 for an illustration.

Selecting the optimal weights for all classes can often become difficult. Applying a simple **inverse frequency** might not work well all the time. The **focal loss** can sometimes help but can potentially result in weighting down all well-classified examples. A third option is to use **sampling**. Check out figure 2.14 for an illustration.

Consider two classes, blue and orange, where blue has far more samples than orange. There are two *pre-processing* techniques that can help us tackle the *imbalance* problem! Keep on reading to find out more!

According to the **undersampling** technique, we select only some examples from the *majority* class - in particular as many examples as the *minority* class has - such that the *probability distribution* of both classes is equal.

According to the **oversampling** technique, we generate a few copies of the *minority* class - in particular as many examples as the *majority* class has - such that the *probability distribution* of both classes is equal.

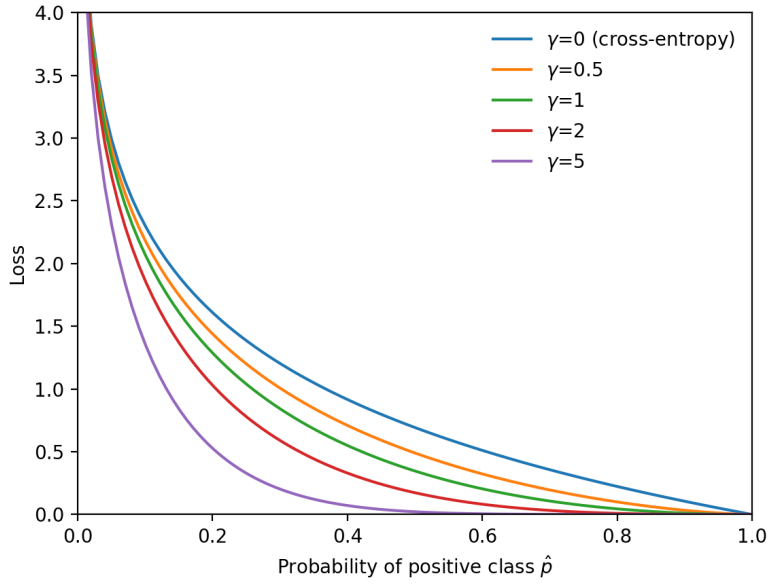

FIGURE 2.13: Focal loss distribution for different γ.

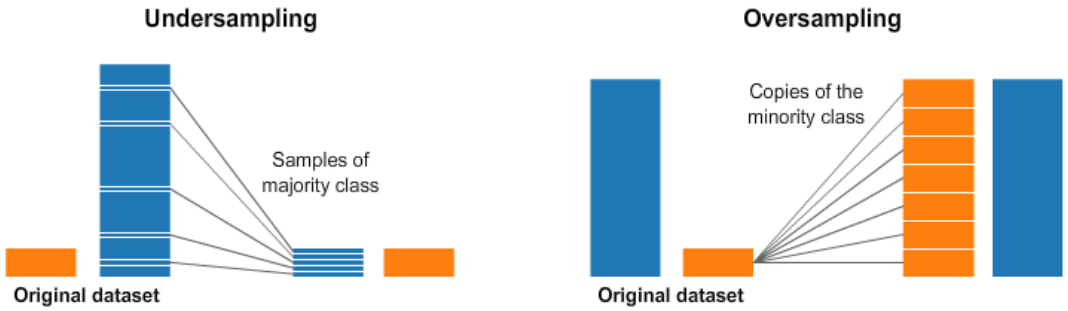

FIGURE 2.14: Undersampling and oversampling.

2.4 Advanced Concepts in Deep Learning

Training a deep *NN* is almost never an easy task. Below are some of the challenges we may encounter during that journey:

- We may face either of the *vanishing* and *exploding* gradients problems. This happens when the gradients are getting smaller and smaller or bigger and bigger, which eventually results in making lower layers very *hard* to train.
- We may not have enough training *samples* for such a large network, or it might be too *expensive* to label new ones.
- The training process may be deadly *slow*.
- A model with millions of parameters runs the risk of *overfitting*, particularly when there are not enough training *samples* or when the available samples are too *noisy*.

In this section, we will go through each of these challenges and present some *techniques* to overcome them. This will enable us to train some very deep *NNs*. So let's get started!

2.4.1 Vanishing and Exploding Gradients Problems

As mentioned earlier in this chapter, the **backpropagation** algorithm spreads the error gradient from the output layer all the way to the input layer. Once the algorithm has computed the gradient of the *loss function* w.r.t. each parameter in the network, it uses these gradients to adjust each parameter with a *GD* step.

However, the gradients often get smaller and smaller as the algorithm moves down to the lower layers. As a result, the weights of the lower layers are left almost intact, and training never converges toward the optimal solution. We call this the *vanishing gradients* problem. At other times, the opposite may happen: the gradients get bigger and bigger until layers get extremely large weight updates, and the algorithm diverges away from the optimal solution. We call this the *exploding gradients* problem. More generally, deep *NNs* suffer from unstable gradients; different layers may learn at entirely different speeds.

This contradictory behavior was experimentally observed a long time ago, and it was one of the reasons deep *NNs* were nearly forgotten in the early 2000s. It was not clear what made the gradients to be so sensitive when training a deep *NN*, but some light was thrown around 2010 when it was shown that, with the *logistic sigmoid* function and the *weight initialization* scheme, the variance of the outputs of each layer is much higher than the variance of its inputs.

2.4.2 Batch Normalization

Although using a better *kernel initializer* and *activation function* can considerably minimize the risk associated with both the *vanishing* and *exploding* gradients at the beginning of training, no one can promise that neither of them will not come back at a later stage of the process.

Fortunately, the *BN* technique, which was introduced around 2015, can address the aforementioned problem in most cases. It does so by adding an operation in the model either *before* or *after* the activation function of each hidden layer. More specifically: first, it *zero-centers* and *normalizes* each input, and then it *scales* and *shifts* the result. The first part should be obvious by now. What the second part does is basically to let the model learn the optimal values for *scale* and *mean* of each of the layer's inputs.

The team of researchers behind *BN* showed that this operation could indeed improve a variety of deep *NNs*. Their efforts led to a significant improvement in the *ImageNet* classification task. The main features of *BN* can be summarised as follows:

- The vanishing gradients problem is *reduced*.
- The networks are less *sensitive* to the weight initialization.
- The learning process is *accelerated* by the use of larger learning rates.
- The need for regularization techniques is reduced since *BN* acts like a *regularizer*.

On the negative side, *BN* adds some complexity to the model. Moreover, the *NN* makes slower predictions due to the extra calculations required at each layer. Fortunately, it's usually possible to combine the *BN* layer with the previous layer, after training, so to avoid the runtime penalty.

2.4.3 Faster Optimizers

Training a very large and deep *NN* can become extremely slow. We have already discussed a few different techniques to accelerate training. Another significant speed boost can come from the use of a faster *optimizer*. In this section, we will present the most popular alternatives to *GD*.

Imagine a bowling ball going down a soft slope on a smooth surface. The ball will start out slowly but will pick up momentum quickly and will eventually reach a maximum speed. This is essentially the main idea behind **momentum optimization**. On the contrary, classic **gradient descent** will simply take small, steady steps down the slope, and hence the algorithm will take significantly more time to reach the bottom of the valley.

At each iteration, **momentum optimization** subtracts the local gradient from the momentum vector, multiplies the vector by the learning rate hyperparameter η, and finally updates the weights by adding this vector. In order to prevent the momentum from growing too large, the algorithm introduces the momentum hyperparameter β, with a typical value of around *0.9*.

Going back to the bowling ball problem again, **gradient descent** starts by quickly going down the steepest slope, although not directly toward the global optimum, and then very gently reaches the bottom of the valley. In contrast, **AdaGrad** corrects its direction earlier to point a bit more toward the global optimum. In short, **AdaGrad** decays the *learning rate*, but it does so faster for steep dimensions than for dimensions with gentle slopes (see figure 2.15). Another advantage is that this algorithm requires significantly less tuning of the learning rate hyperparameter η.

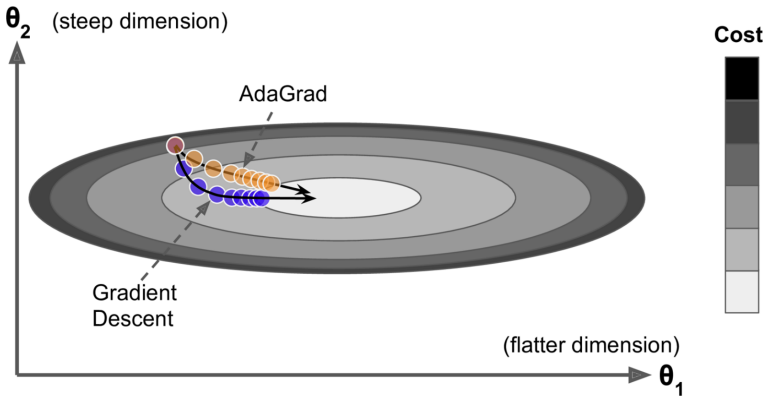

FIGURE 2.15: AdaGrad vs. gradient descent.

Despite all the improvements, **AdaGrad** runs the risk of slowing down much faster and not converging to the global optimum. **RMSProp** overcomes this limitation by accumulating only the gradients from the most recent iterations rather than all the gradients since the beginning of training. In order to accomplish this, the algorithm introduces the decay rate hyperparameter β, with a typical value of around *0.9*.

Another very popular algorithm called **Adam**, combines the ideas of **momentum optimization** and **RMSProp**. In short, this algorithm keeps track of an exponentially decaying average of past gradients as well as of past squared gradients. Since it's an

adaptive *learning rate* algorithm, it requires less tuning for the learning rate hyperparameter η, with a typical value of around *0.001*, making this algorithm even easier to use than **gradient descent**.

Finding a reasonable *learning rate* is very important and challenging at the same time. If we set it too high, training may *diverge*. If we set it too low, training will eventually *converge* to optimum, but very possible, it will take a long, long time. If we set it a little too high, it will make good progress at the beginning, but very possible, it will end up *jumping around* the optimum and never really *converge*. In reality, we will always face limitations in computational resources, which will force us to interrupt training before it has wholly converged, yielding a sub-optimal solution (see figure 2.16).

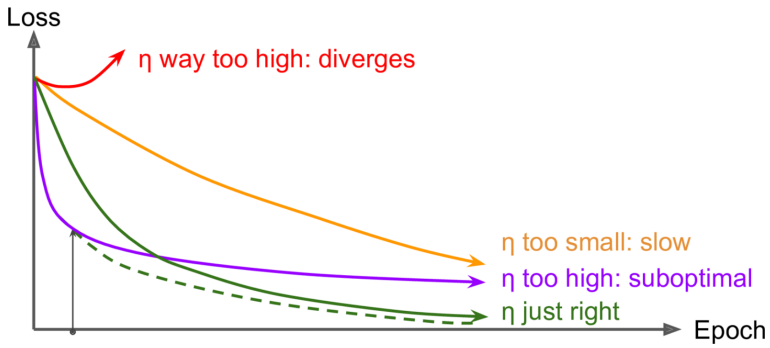

FIGURE 2.16: Learning curves for various learning rates η.

An easy way to find an acceptable *learning rate* is by training the network for a few hundred iterations, exponentially increasing the *learning rate* from small to large values, and then looking at the *learning curve* and selecting a value marginally smaller than the one at which the *learning curve* starts coming back. The final step involves the re-initialization and the re-training of the network based on the chosen *learning rate*.

In practice, however, we can always do better than using the optimal constant *learning rate* as there are far better ways to reach a sub-optimal solution much faster. For example, we can start with a large *learning rate* and then reduce it once training stops making fast progress. An alternative idea would be to start with a small *learning rate*, then increase it, and then drop it again. These strategies are called **learning schedules** with the most commonly used ones listed below:

- Power scheduling.
- Exponential scheduling.
- Piecewise constant scheduling.
- Performance scheduling.
- 1cycle scheduling.

2.4.4 Overfitting and Underfitting

One of the biggest challenges in *ML* is finding the right balance between **optimization** and **generalization**. The former is about adjusting the model to get the best performance possible on the training data, whereas the latter is about how well the trained model is performing on data it has never seen before. Although our goal is always

to get good generalization, in practice, there is no way to control it; all we can do is adjusting our model based on the training data.

When training begins, *optimization* and *generalization* are going hand-in-hand: i.e., when the loss on training data is getting lower, so does the loss on test data. At this point, our model is said to be **underfit**, which essentially means that the network has not yet modeled all relevant patterns in the training data. However, after a few iterations, the model is said to be **overfit**, which essentially means that the network begins to learn patterns that are specific to the training data but not very relevant to the test data.

The moment we notice *overfitting* or *underfitting*, we have to decide whether it's more reasonable to spend time pre-processing the data and re-tuning the network or go out and collect more data. The last thing we want is to spend a significant amount of time and money working in one direction only to find out it hardly improved the network performance.

A common misbelief among *ML* beginners is that throwing more data can always improve the performance of the model. Unfortunately, collecting and labeling more data is not always an option in reality. Also, depending on the problem, it could be very expensive or time-consuming to collect and label new data.

If acquiring more data is not an option, the next best solution is to control the quantity of information or to add constraints on what information our model is permitted to store. If a network can only learn a small number of patterns, the optimization process will force it to concentrate on the most important patterns, which have a higher likelihood of generalizing well.

As we discussed earlier, one way to escape *overfitting* is to get more training data. However, this is not always a feasible solution. An alternative solution is to *augment* the training data by generating new instances through some special type of transformations called **data augmentations**. This is a very popular technique that gives the learning algorithm more training data and ultimately reduces *overfitting*. There are various types of transformations that we can use, including, but not limited to, **flipping**, **rotating**, **scaling**, **zooming**, **lighting**, etc. Figure 2.17 illustrates some of these transformations applied to an image of the *6* digit.

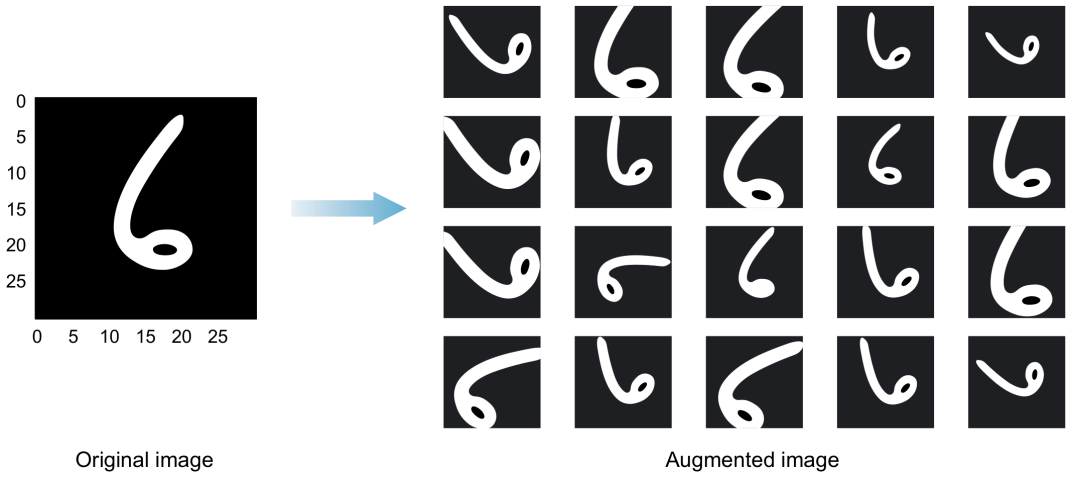

FIGURE 2.17: Transformations applied to a MNIST image.

Another way to prevent *overfitting* is to reduce the **extent** of the model. As expected, a model with more parameters has a bigger memory capacity, making the learning of mappings between samples and targets much easier, but with the cost of poor *generalization*. In contrast, a model with a smaller memory capacity will not be able to learn the mappings as easily, forcing it to learn only compact representations. At the same time, a model should have enough parameters to avoid *underfitting*. As we can see, there is always a trade-off between too much capacity and not enough capacity.

Sadly, there is no secret sauce when it comes to finding the ideal **number** of layers or the ideal **size** for each layer. Instead, we have to evaluate a number of different *architectures* in order to decide the most appropriate one for our training data. It's very common to start with relatively few layers and parameters and then add new layers or increase their size until we see decreasing *loss* returns on the validation set.

Intuitively, complex models are more likely to overfit than simple ones. Within this framework, a simple model is a model in which the *distribution* of parameter values has less *entropy*. A common technique to ease *overfitting* is to set constraints on the complexity of the network by pushing its *weights* to take smaller values and hence make their *distribution* more *regular*. This is the so-called **weight regularization**, which comes into two flavors as shown below, and works by adding to the *loss function* a cost associated with having large *weights*.

- **L1 regularization**: the cost added is proportional to the **absolute** value of the weight coefficients.
- **L2 regularization**: the cost added is proportional to the **square** value of the weight coefficients.

Finally, let's briefly talk about one of the most commonly used layers to prevent *overfitting* called **dropout**. It consists of randomly setting to *zero* a percentage of neurons during training. This percentage, called **dropout rate**, is identified as a hyperparameter with values ranging from *0.2* to *0.5* (see figure 2.18).

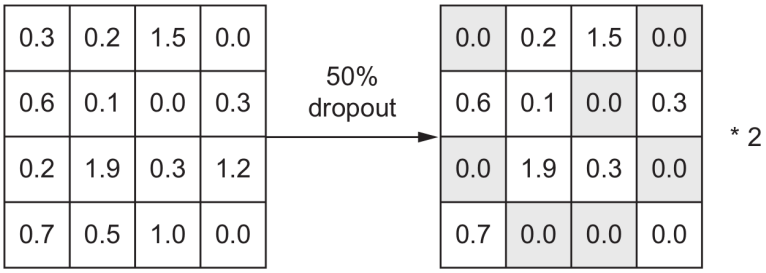

FIGURE 2.18: Dropout applied to an activation matrix.

By *zeroing*, we mean these neurons are not considered during a particular **forward** or **backward** pass. It may seem counterintuitive to throw away a connection in our network, but as the network is getting trained, some nodes can dominate others or end up making large mistakes, and **dropout** gives us a way to *balance* our network so that every node works *equally* towards the same goal, and if one makes a mistake, it will not dominate the behavior of the model as a whole. We can think of **dropout** as a technique that makes a network *resilient*; it makes all the nodes work well as a *team* by making sure no node is too **weak** or too **strong**.

Chapter 3

Deep Learning in Computer Vision

3.1 Introduction

CV is the most rapidly growing *CS* field nowadays, thanks to the immense advances in *AI* and *DL* that took place a few years ago. The most exciting thing of all is how rapid advances in research enable new applications to be built every day and across different domains. Such applications were impossible to be developed just a few years ago.

Although this chapter is dedicated to *CV*, many of the concepts here are also related to *DL*. For example, we will have the chance to talk about the most popular architectures, namely, **ANNs**, **CNNs**, and **RNNs**.

The core concept in *AI* systems is to perceive the environment and take actions based on these perceptions. *CV* deals with the visual perception part. It's the science of perceiving and understanding the world through images and videos by constructing a physical model of the world so that *AI* systems can act appropriately.

Let's dive deeper into each of the concepts involved to better understand how they work.

3.1.1 Visual Perception

At its most basic form, *visual perception* is the act of observing patterns and objects through images. Let's take autonomous vehicles as an example. Under this context, *visual perception* is about understanding the surrounding objects and their specific details, such as pedestrians, lane centering, traffic signs, etc. That is why the word **perception** was added to the definition. We are not just looking to capture the surroundings. We are trying to build systems that can really understand the environment through **visual** inputs.

3.1.2 Vision Systems

Up until recently, the terms *image processing* and *computer vision* were used interchangeably. But as you can imagine, this is not very accurate; having machines **processing** an image is entirely different from **understanding** what is presented in an image. Image processing became part of a bigger, more complex system that aims at understanding the content of an image, not just processing it.

Fundamentally, *vision systems* are the same for humans, animals, insects, and most living organisms. They consist of a sensor (or **eye**) to capture the image and a unit (or **brain**) to process and interpret the image. The system outputs a prediction of the image components based on the information extracted from the image (see figure 3.1).

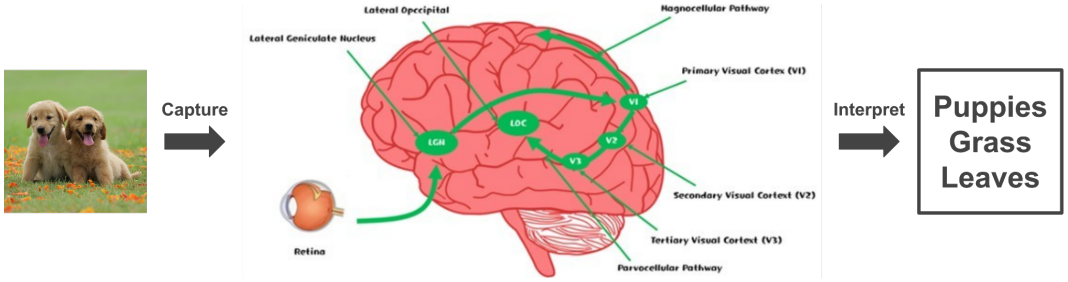

FIGURE 3.1: Human vision system.

Now let's examine how the *human vision* system works. Suppose we want to interpret the image of the dogs above. We look at it and we directly understand that the image consists of a few dogs. It comes pretty naturally to us to detect and classify objects in this image because we have been trained over the years to identify dogs. We can train our brains to identify almost anything. Same with computers. We can teach machines to learn and identify objects, but humans are much more intuitive than machines. It takes just a *few* image samples for humans to learn to identify objects, but it takes *thousands* or even *millions* of image samples to learn to identify objects for machines.

The *human vision* system has inspired scientists in recent years to extend this visual ability to machines. So, in order to mimic the *human vision* system, we need the same two components, namely, a sensing device to mimic the function of the **eyes** in capturing the image and a sophisticated algorithm to mimic the function of the **brain** in interpreting the content (see figure 3.2).

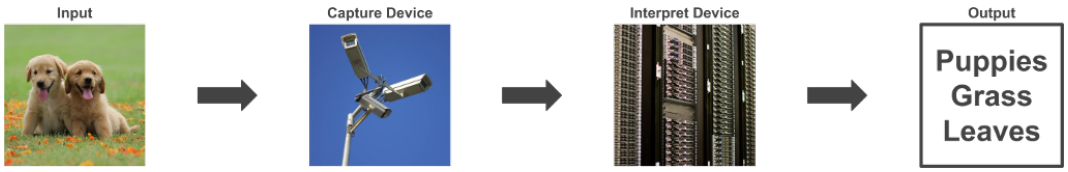

FIGURE 3.2: Computer vision system.

3.1.3 Sensing vs. Interpreting

As we already mentioned, the *vision systems* are designed to fulfill two specific tasks, and in order to perform them, they rely on two specialized components.

The first one is the **sensing** device, the aim of which is to *capture* the surroundings of the specific environment. It goes without saying that this first component is considered the **eye** of the vision system. Whether it's a camera, a radar, an X-ray, or a combination of them, the goal is always to provide the environment's full scene to accomplish the task at hand. Each of these devices is used for a specialized task. As a result, the first step in designing end-to-end vision systems is always to choose the right device for the right problem.

The second one is the **interpreting** device, the aim of which is to take the output from the first component and *learn* to identify objects of interest in it. It goes without saying that this second component is considered the **brain** of the vision system. It is most commonly implemented in the form of *ANNs*, which were invented as part of the scientists' efforts to build an artificial brain. Inspired by how the human brain works, scientists reverse-engineered the central nervous system to gain useful insights and ultimately build an artificial brain.

One could clearly say that there is an analogy between biological neurons and artificial systems. Both contain a main processing element called *neuron*, an *input* signal $X_1, X_2, ..., X_n$, and, of course, an *output* value (see figure 3.3).

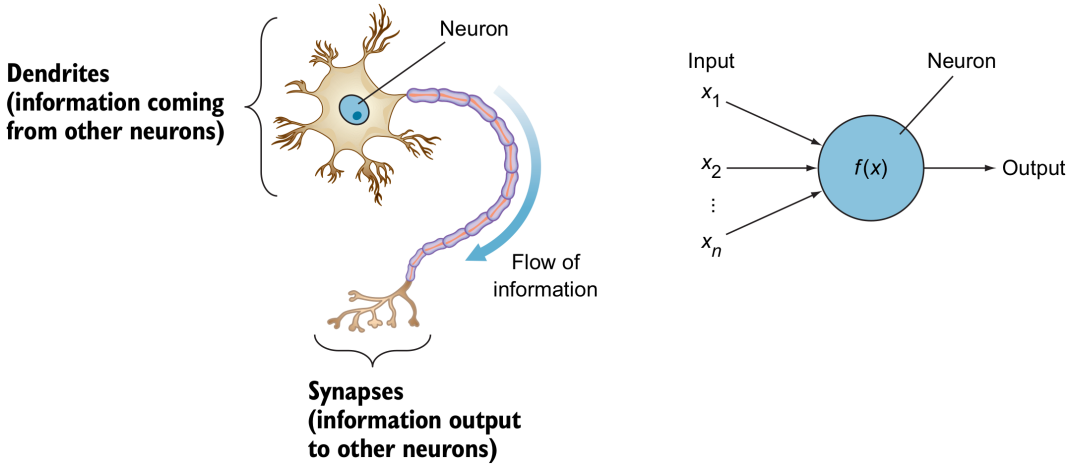

FIGURE 3.3: Biological vs. artificial neuron.

The *learning behavior* of **biological** neurons inspired scientists to create an **artificial** network of neurons that are connected to each other. Imitating how information is processed in the human brain, each individual artificial neuron fires a signal to all neurons that it's connected to when enough of its input signals are activated. Thus, neurons have a very simple mechanism on the individual level, but having many of these neurons stacked in layers, with each of them connected to thousands of other neurons, ultimately leads to a *learning behavior* (see figure 3.4).

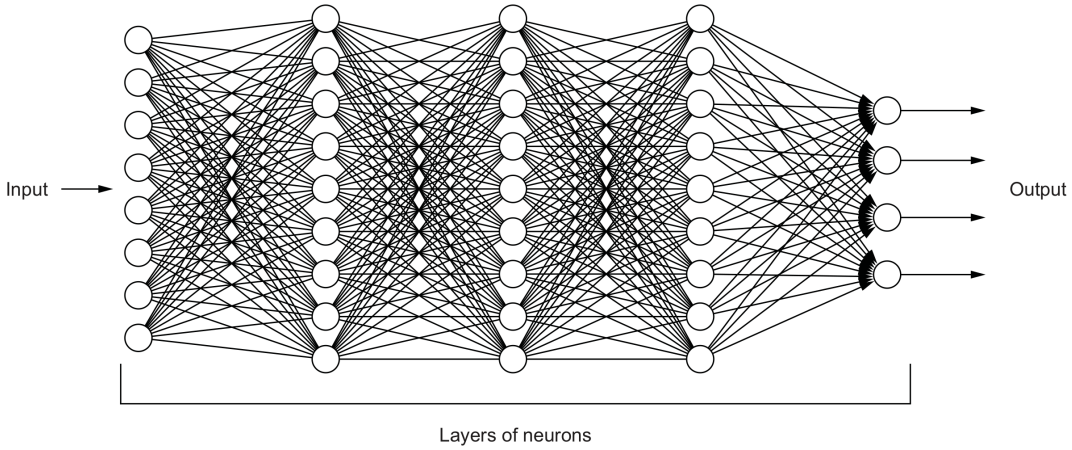

FIGURE 3.4: Artificial neural network.

Building a multi-layer *NN* is called *DL*. *DL* models learn representations through a sequence of data transformations that go several layers deep. In the next chapter, we will study the most popular *DL* architectures, namely, **ANNs**, **CNNs**, and **RNNs**, and we will present how they are used in different *DL* applications.

3.1.4 Machines vs. Humans

Some people may wonder if *ML* can achieve better performance than the human brain. Well, if they had asked this question ten years ago, then, most probably, the answer would be **no**, *machines* cannot surpass *humans*. But let's take a look at the following scenarios:

- Suppose we were given a book of 10k dog images classifying their breeds, and we were asked to learn the properties of each breed. How long would it take us to study the different 130 breeds in 10k images? And if we were given a test of 100 dog images and asked to label them based on what we learned, out of the hundred, how many will we get right? Well, a *NN* that is trained in a couple of hours, can achieve more than *95%* accuracy.
- But *NN* can contribute to arts too. For example, a *NN* can study the patterns in the strokes, colors, and shading of a particular artwork and then transfer the style from the original image into a new one, creating a genuine piece of art within seconds.

It turns out that the recent *AI* and *DL* advances have allowed machines to surpass human visual abilities in many image classification and object detection tasks. At the same time, their capabilities have been expanded to many other fields as well. But, do not take our word for it; let's revisit this question after the next section, where we review some of the most popular *CV* applications using *DL* technology.

3.1.5 Sample Applications

Computers started to recognize human faces in images decades ago, but only in recent years have *AI* systems started rivaling computers' ability to classify objects in photos and videos. Thanks to the dramatic evolution, both in computational power and available data, *AI* and *DL* have achieved superhuman performance in many complex visual perception tasks, including, but not limited to, **image search**, **image captioning**, **image classification**, and **object detection**.

Although the focus of this work is on visual applications, deep *NNs* can be used in a variety of tasks. Of course, due to time constraints, we won't list all possible applications here; this would require an entire thesis. Instead, we will give a bird's eye view on some of the most important *DL* concepts from the aspect of **medical image diagnosis**.

Image classification aims at assigning a *label* to an *image* from a pre-defined set of categories. *CNN* is the type of *NN* that truly shines in processing and classifying images in many different applications. The lung cancer diagnosis is such a growing problem (see figure 3.5). Several *CV* companies have decided to tackle this challenge using *DL* technology. The main reason lung cancer is very dangerous is that physicians usually diagnose this in the mid or late stages. When examining *CT* scans for lung

cancer, doctors typically use their eyes to search for small nodules in their patients' lungs. At the early stages, the nodules are usually tiny and harder to spot.

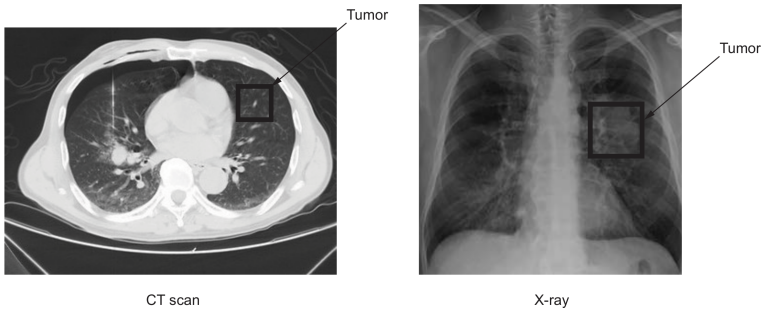

FIGURE 3.5: Lung cancer diagnosis.

Almost every lung cancer starts as a small nodule. In addition, it always appears in a wide variety of shapes. Unfortunately, it takes doctors years to learn all these different shapes. Doctors are very good at identifying medium and large nodules (i.e., around 6-10 millimeters). But the smaller ones (i.e., like 4 mm or smaller) are very hard to get spotted. Deep *NNs*, especially *CNNs*, can learn these features automatically from *X-ray* or *CT* scans. Most importantly, they can detect small nodules much earlier, hence way before the start becoming deadly.

3.2 Deep Convolutional Neural Networks

This chapter introduces *CNNs*, a class of deep *NNs*, used almost universally in *CV* applications. We'll learn to apply them to image classification problems, particularly those involving small training datasets, which is the most common use case in academia. On the contrary, big tech companies usually work on much bigger and proprietary datasets.

3.2.1 Quick Introduction

Let's dive deeper into the theory of what *CNNs* are and why they have been so successful at *CV* tasks.

Gist 3.1 shows a basic *CNN*, which is essentially a stack of **Conv2D** and **MaxPooling2D** layers. We'll see in a minute exactly what they do and how they work.

```
>> from keras import layers
>> from keras import models

>> model = models.Sequential()

>> model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape
    =(28, 28, 1)))
>> model.add(layers.MaxPooling2D((2, 2)))
>> model.add(layers.Conv2D(64, (3, 3), activation='relu'))
>> model.add(layers.MaxPooling2D((2, 2)))
>> model.add(layers.Conv2D(64, (3, 3), activation='relu'))

>> model.add(layers.Flatten())
>> model.add(layers.Dense(64, activation='relu'))
```

```
>> model.add(layers.Dense(10, activation='softmax'))

>> model.summary()
```

GIST 3.1: Basic convolutional neural network implementation.

Usually, *CNNs* are fed with input tensors of shape *(image_width, image_height, image_channels)*. For example, here, we tell the network to process inputs of size *(28, 28, 1)*, which is the required format for the *MNIST* dataset. This can be easily done by setting argument *input_shape=(28, 28, 1)* in the first layer.

As we can see, every *Conv2D* and *MaxPooling2D* layer's output is a *3D* tensor of shape *(image_width, image_height, image_channels)*. Moreover, both **width** and **height** shrink as we go deeper into the network. On the contrary, **channels** is controlled by the first input argument of the *Conv2D* layers (usually of size 32 or 64).

The next step is to feed the final output tensor (of shape *(3, 3, 64)*) into a densely-connected classifier. However, the problem is that the classifier requires a *1D* vector, whereas the current output is a *3D* tensor. Therefore, we need to flatten the *3D* output into a *1D* one and then add some densely-connected layers on top.

As we can see, the *(3, 3, 64)* outputs are flattened into vectors of shape *(576,)* before going through the two densely-connected layers.

For the *MNIST* dataset, a ten-way classification is required. We can do so using a final layer with **ten** outputs and a **softmax** activation function. Gist 3.2 shows the architecture of the underlying network:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)  (None, 13, 13, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)  (None, 5, 5, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                36928
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

GIST 3.2: Basic convolutional neural network summary.

Without further ado, let's discuss the underlying network's main components (see figure 3.6). The corresponding text representation goes like this: *INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => SOFTMAX*. Note that both **RELU** and **SOFTMAX** are not really standalone layers but rather the previous layers' activation functions. They are presented like that here to highlight that the network designer used the **RELU** activation function in the *CONV* layers and the **SOFTMAX** activation function in the *FC* layer. Furthermore, it worth mentioning that although

the network contains two *CONV* layers and one *FC* layer, in practice, we can add as many *CONV* and *FC* layers as we see fit. Finally, recall that the *CONV* layers are used for the **feature extraction**, whereas the *FC* layers are used for the actual **classification**.

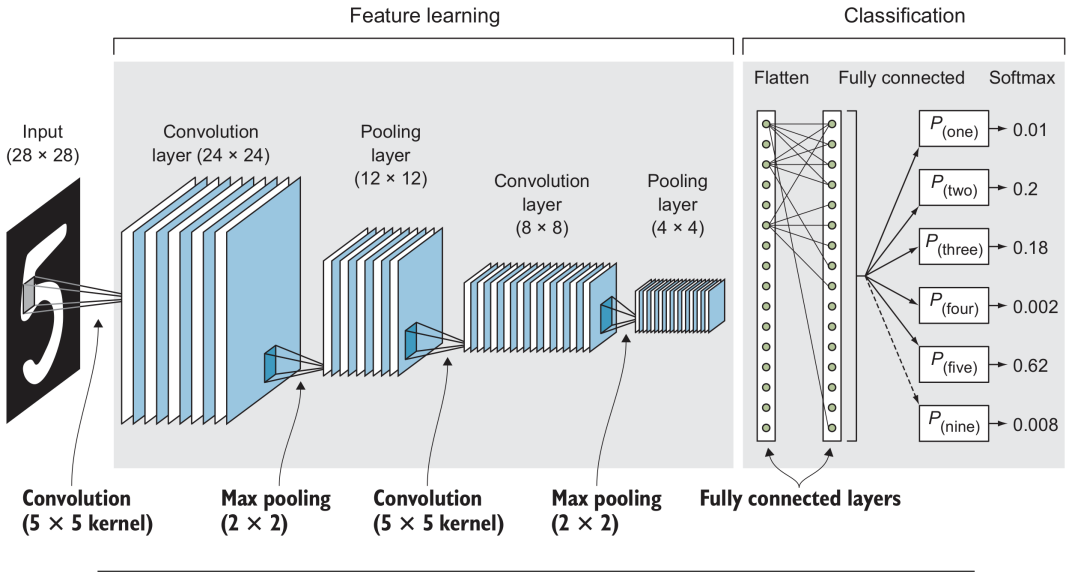

FIGURE 3.6: Basic convolutional neural network components.

There are three main types of layers that we will see in almost every *CNN*:

- Convolutional layer.
- Pooling layer.
- Dense layer.

Now that we saw the full architecture let's dive deeper into each layer type separately to understand better how they work. Then, at the end of this section, we will put them all back together.

3.2.2 Convolutional Layer

Convolutional layers are the building blocks of *CNNs*. They act like a feature finder window that slides over the image pixel-by-pixel to extract meaningful features that identify objects in the image.

In *functional analysis*, convolution is the operation of two functions to produce a third modified function. In the context of *computer vision*, the first function is the input image, whereas the second function is the convolutional filter. We can think of it as a series of matrix multiplications to produce a modified image with new pixel values.

Let's zoom into the first convolutional layer to see how it processes an image (see figure 3.7). By sliding the convolutional filter over the input image, the network breaks the image into little chunks and processes them individually to assemble the modified image called **feature map**.

Now that we have the big picture, let's define some key terms (see figure 3.8):

- The small 3×3 matrix is the *convolutional filter*, a.k.a. **kernel**.

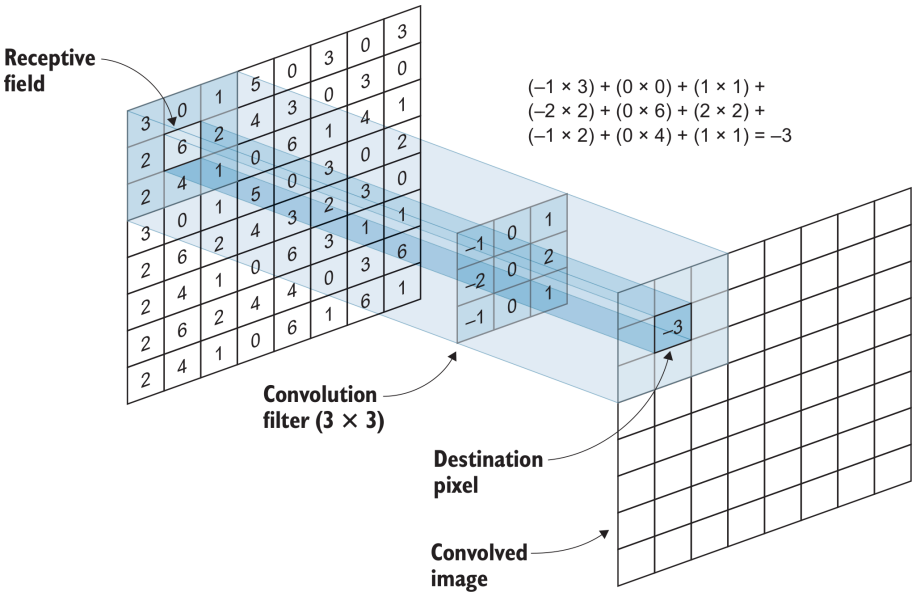

FIGURE 3.7: Zoom in to a convolutional layer.

- The **kernel** slides over the original image *pixel-by-pixel* and does some *matrix multiplications* to get the new convolved image's values at the next layer.

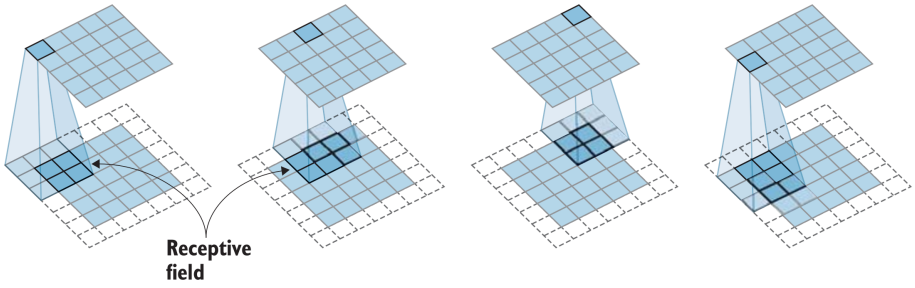

FIGURE 3.8: Zoom in to a convolutional filter.

The **convolutional matrix** is for *CNNs* what the **connection weights** are for *MLPs*. That means it is *randomly initialized* at the beginning, and its values are *learned* by the network later on during training.

The **kernel** is a matrix of weights that *slides* over the image to *extract* features. The **kernel size** here refers to the dimensions of the matrix (see figure 3.9).

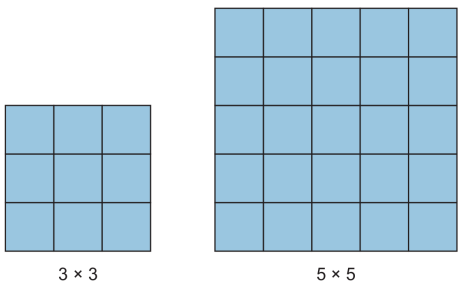

FIGURE 3.9: Common kernel sizes.

The **kernel size** is one of the *hyper-parameters* that needs to be set when building a convolutional layer. Like most *hyper-parameters*, there is no right answer that fits all problems. The rule of thumb is that *smaller* filters capture very small details in the image, whereas *bigger* filters miss these details.

Recall that these filters contain the weights that will be *learned* by the network during training. Theoretically, the bigger the **kernel size** is, the deeper the network, and hence, the better it learns. Unfortunately, this comes with higher computational *complexity* as well as a high chance of *overfitting*.

Kernels are almost always of *square* shape, with dimensions ranging from 2×2 to 5×5. Technically, we can use much bigger sizes, but very possible, this will cause the loss of valuable information from the image.

The fundamental difference between *convolutional layers* and *densely-connected layers* is that the former learn **local** patterns in their input feature space, whereas the latter learn **global** ones. In the case of images, local patterns are found in small 2D windows (e.g., of 3×3 size) of the inputs (see figure 3.10).

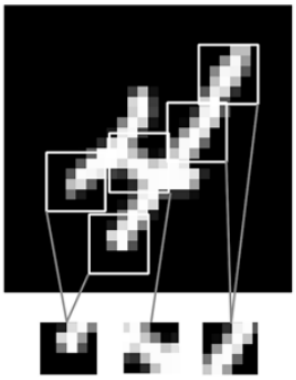

FIGURE 3.10: Convolutional layers learn local patterns.

This key feature gives *CNNs* two interesting properties (see figure 3.11):

- **Translation-invariant pattern learning**: After learning a certain pattern in the upper-left corner of an image, CNN can recognize it anywhere. A densely-connected network will have to learn the pattern again if it appears at another location. This property makes CNNs very efficient when processing images (because the visual world is fundamentally *translation invariant*).
- **Hierarchical-spatial pattern learning**: The first convolutional layer will learn small local patterns such as edges, the second one will learn larger patterns made of the features in the first layer, and so on so forth. This property allows CNNs to learn extremely complex visual concepts very efficiently (because the visual world is fundamentally *spatially hierarchical*).

Convolutions work with *3D* tensors, called *feature maps*, consisting of two spatial and one depth axes (called *width*, *height*, and *channels*, respectively). In color images, the depth is always *three* because those images are based on the *RGB* color model. On the contrary, in gray-scale images, the depth is always *one* simply because those images have no color. The convolution operation extracts chunks from its input *feature map* and applies the same transformation to all of these chunks, producing an output *feature map*. This map is still a *3D* tensor as it has *width*, *height*, and *depth*. However, the

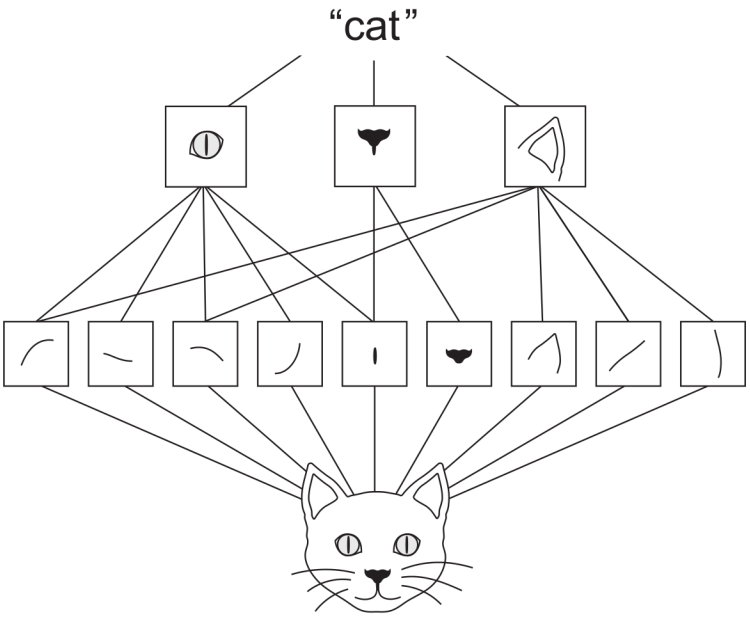

FIGURE 3.11: CNNs learn spatial hierarchies of patterns.

latter can be arbitrary because the different channels no longer stand for specific *RGB* colors. Instead, they stand for *filters* that encode particular aspects of the input data, e.g., a single filter could encode the concept of presence or absence of dogs in images.

Going back to the *MNIST* example, the first convolutional layer reads a *feature map* of size *(28, 28, 1)* and writes a *feature map* of size *(26, 26, 32)*. Each of the 32 output channels contains a 26×26 array of values, which is a response map of the filter over the input, specifying the response of that filter pattern at different locations in the input (see figure 3.12). That is what the word *feature map* means: every dimension in the depth axis is a filter, and the *2D* tensor is a spatial map of the response of this filter over the input.

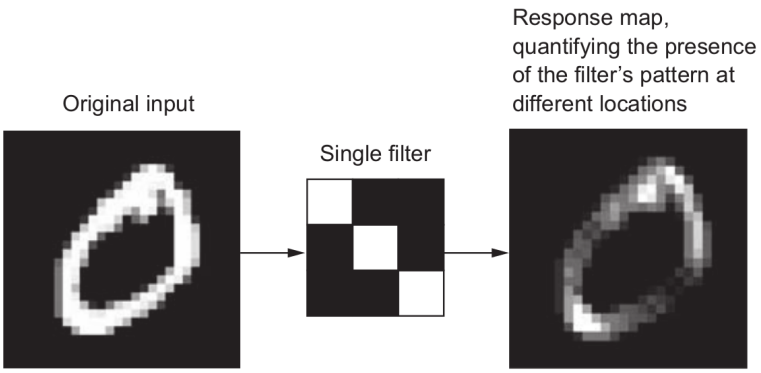

FIGURE 3.12: From input image to response map.

Two key parameters specify convolutions:

- The **size** of the chunks extracted from the input images; the most typical sizes are 3×3 and 5×5.

- The **depth** of the output feature map computed by the convolution; the most typical sizes are 32 and 64.

Convolutions work by sliding the *kernel* over the *3D* input *feature map*, stopping at every possible spot, and extracting the *3D* chunk of the surrounding features. Each of these chunks is then transformed via the *convolution kernel* into a *1D* vector. All of these vectors are then spatially recompiled into a *3D* output *feature map*. Every spatial spot in the output *feature map* corresponds to the same spatial spot in the input *feature map*. For instance, the lower-right corner of the output contains information about the lower-right corner of the input. The full process is illustrated in figure 3.13).

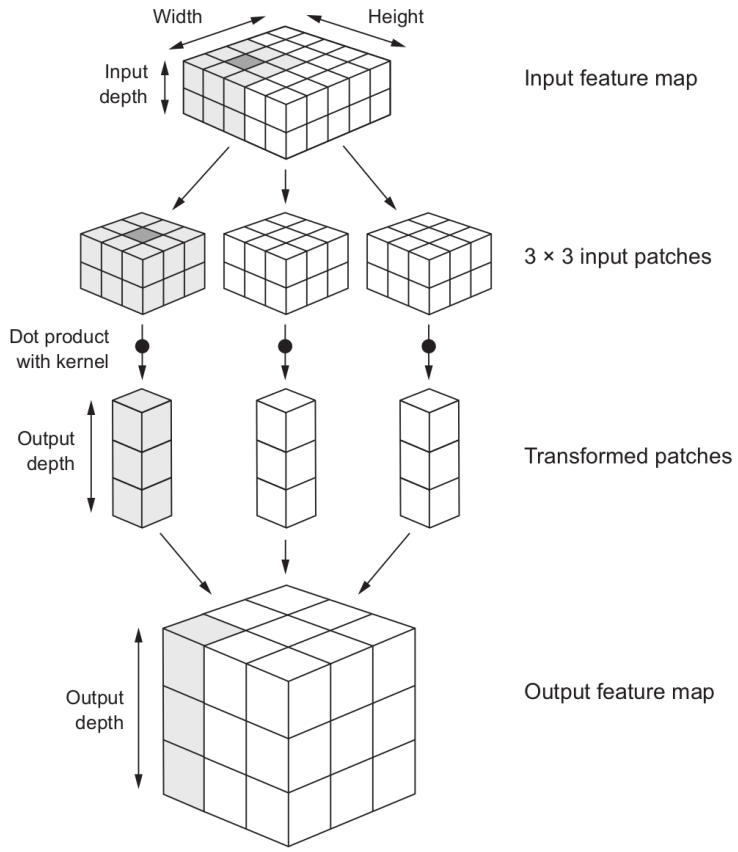

FIGURE 3.13: Multi channel 2D convolution.

We will not go into details here but bear in mind that the output spatial axes may differ from the input ones. This may happen for two reasons:

- The border effect.
- The use of strides.

3.2.3 Pooling Layer

Adding more convolutional layers increases the *depth* of the output layer, which leads to an increase in the number of *parameters* that the network needs to learn. The addition of hundreds of convolutional layers will generate a huge number of parameters. The increase in the network dimensionality will increase the computational *time* and

complexity in the learning process. This is exactly when the pooling layer comes in handy. The *pooling* technique reduces the network's size by reducing the number of parameters passed to the next layer. The pooling operation resizes its input by applying a summary statistical function, such as *minimum*, *maximum*, *average*, etc., to reduce the overall number of parameters passed on to the next layer.

The pooling layer's goal is to *downsample* the feature maps produced by the convolutional layer into a smaller number of parameters to reduce the computational *time* and *complexity*. In practice, it's very common to add a pooling layer after every convolutional layer in *CNNs* (see figure 3.14).

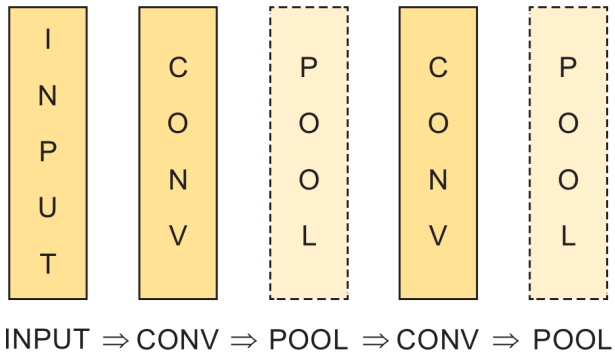

FIGURE 3.14: Pooling layer.

As we can see from the previous example, pooling layers reduce the *dimensionality* of the convolutional layers. This task is very important because complex *CNNs* contain many convolutional layers, each of which with tens or hundreds of kernels. Since each kernel contains the weights that the network learns, this can get out of control very quickly, and the *dimensionality* of the convolutional layers can get very large. Adding pooling layers will help us keep the important features only and pass them along to the next layer, while shrinking the image *dimensionality*. Think of it as an image compression process: we can reduce the image resolution while keeping its most important features (see figure 3.15).

FIGURE 3.15: Dimensionality reduction.

In this example, the size of feature maps is cut in *half* after each *MaxPooling2D* layer. More specifically, before the first *MaxPooling2D* layer, the feature maps are 26×26; however, the max pooling operation cut them in half, i.e., 13×13. That's precisely the max pooling operation's role: to aggressively *downsample* feature maps much like *strided convolutions*.

The max pooling operation consists of *extracting* windows from the input feature maps and *outputting* each channel's max value. It's conceptually similar to convolution, except that instead of transforming local chunks via a *learned* linear transformation, they're transformed via a *static* max tensor operation. A big difference from convolution is that max pooling is usually performed with 2×2 windows and stride two, whereas convolution is usually performed with 3×3 windows and no stride at all.

Simply put, the reason to use *downsampling* is to decrease the number of *feature-map coefficients* to process as well as to produce *spatial-filter hierarchies* by making successive convolutional layers look at increasingly large windows.

It worth pointing out that max pooling isn't the only way to achieve *downsampling*. As we discussed previously, we can also use strides in the earlier convolutional layer or use average pooling where each local input chunk is transformed by taking the average value of each channel over the chunk. However, in practice, max pooling tends to work better than either of these alternatives. The reason is, features tend to encapsulate the spatial presence of some patterns over the different blocks of the feature map, and it's more informative to look at the maximal presence of different features than at their average presence. To conclude, the most reasonable *subsampling* technique is first to produce *dense maps* of features and then look at the features' *maximal activation* over small chunks. Any other technique will probably cause the miss or the reduction of feature-critical information.

3.2.4 Dense Layer

After going the image through the feature learning process using convolutional layers and pooling layers, we have extracted all the features and have put them in a long pipe. The next step is to use the extracted features to classify images. One way to achieve this is by using the typical *NN* architecture called *MLP*.

MLPs work great in classification problems. Previously, we used convolutional layers because *MLPs* lose important information when extracting features from images. On the contrary, convolutional layers can process images without huge drawbacks. Now, assuming that we have the features extracted and flatten, we can use regular *MLPs* to classify new images.

We have already discussed the *MLP* architecture in detail in a previous section (see figure 3.16).

To recap, here are the main components of this special type of *NN*:

- Input vector: In order to feed the features pipe of size (5, 5, 40) to the MLP for classification, we need to flatten the former into a vector of size (1000,).
- Hidden layer: Similarly to what we do when building regular MLPs, we need to add one or more fully connected layers, with each of them having one or more neurons.
- Output layer: When our task involves more than two classes, it's adviced to use a softmax activation function with the same number of nodes as the number of classes.

By now, we should be familiar with the three main types of layers in *CNNs*, namely, **convolutional**, **pooling**, **dense**, as well as how to build small networks to solve simple

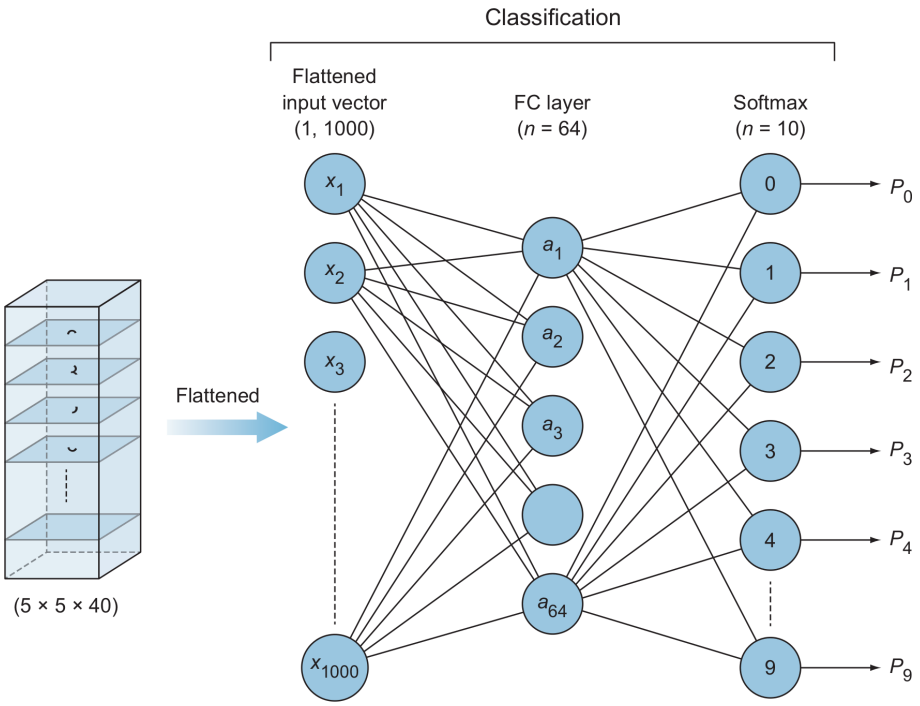

FIGURE 3.16: Multi layer perceptron.

problems such as the *MNIST* digit classification task. Now let's switch gears and talk about the two most important *CV* concepts, namely, **data preparation** and **transfer learning**.

3.3 Data Preparation

Preparing image data for *CNNs* is a challenging task. It involves *scaling* the pixel values and using *augmentation* techniques during both *training* and *evaluation* steps.

This section presents some best practices around image preparation in classification tasks. At the end of this section, we should be able to understand why:

- Images must be centered by subtracting the per-channel mean pixel value calculated on the training set.
- Augmentation during training must involve random rescaling, horizontal flips, perturbations to brightness, contrast, and color, as well as random cropping.
- Augmentation during testing must involve a mixture of multiple rescaling of each image and predictions for multiple different systematic crops of each rescaled version of the image.

Let's get started.

3.3.1 Preprocessing

We should now understand that data must be formatted appropriately into pre-processed floating-point tensors before getting fed into a network. Initially, the data live on a

hard drive as *JPEG* files. The steps for getting those files into a network are more or less as follows:

- Read the image file.
- Decode the JPG content into a grid of RGB pixels.
- Convert the integers into floating-point tensors.
- Rescale the pixel values from [0, 255] to [0, 1].

The process may seem challenging at first, but luckily for us, *Keras* can cope with all of these steps with ease. More specifically, class *ImageDataGenerator* of the *keras.preprocessing.image API* allows us to quickly set up *Python* generators to automatically turn image files on disk into infinite batches of pre-processed tensors. Gist 3.3 shows how this can be accomplished:

```
>> from keras.preprocessing.image import ImageDataGenerator

>> train_datagen = ImageDataGenerator(rescale=1./255)
>> test_datagen = ImageDataGenerator(rescale=1./255)

>> train_generator = train_datagen.flow_from_directory(
  train_dir,
  target_size=(150, 150)
  batch_size=20,
  class_mode='binary'
)

>> validation_generator = test_datagen.flow_from_directory(
  validation_dir,
  target_size=(150, 150),
  batch_size=20,
  class_mode='binary'
)
```

GIST 3.3: Processing the data using batch and image generators.

Now, let's have a look at the output of a generator. The generator yields batches of 150×150 color images with their corresponding labels. Each batch consists of 20 samples, and therefore each iteration generates two objects, i.e., a matrix of size *(20, 150, 150, 3)* and a vector of size *(20,)*. Also, note that the generator runs indefinitely, i.e., it loops endlessly over the image folder. As a result, it is required to break the iteration at some point, i.e., after a certain number of batches has been generated.

Next, we fit the model to the data using the image generator. We do so using the *fit_generator* method, the first argument of which is a *Python* generator that yields batches of inputs and targets endlessly. *Keras* needs to know how many samples to draw before considering an epoch as finished because of the indefinite data generation. Argument *steps_per_epoch* comes to the rescue; after running for *steps_per_epoch* steps, the fitting process moves to the next epoch. For example, here, we have batches of size *20*, so it takes *100 steps_per_epoch* to reach our target of *2,000* samples.

When we use the *fit_generator* method, we can also pass a *validation_data* argument, with two possible types of values: either a different *Python* generator or a tuple of *Numpy* arrays. If the former, then, this will yield validation batches indefinitely. So, like previously, we must specify the *validation_steps* argument, which specifies the number of batches to be drawn for evaluation. Gist 3.4 shows how this can be accomplished:

```
>> history = model.fit_generator(
  train_generator,
  steps_per_epoch=100,
  epochs=30,
  validation_data=validation_generator,
  validation_steps=50
)
```

GIST 3.4: Fitting the model using a batch generator.

In cases where the number of training images is relatively small, *overfitting* will be our number-one concern. We have already discussed a few techniques for dealing with *overfitting*, such as **dropout** and **weight decay**. In the next section, we will present a new one called **data augmentation**, which is inextricably linked to *CV* tasks and commonly used in *CNN* trainings.

3.3.2 Augmentation

The number one reason for *overfitting* is due to having too few training samples, making us unable to build a model that *generalizes* well on new data. If we had unlimited data, our model would be exposed to the whole data distribution, and hence, it would never *overfit*. *Augmentation* is based on the idea of generating artificial samples from existing ones by *augmenting* the samples through a sequence of random transformations that result in genuine-looking samples. The goal is that, at training time, our model will never see the exact same image twice. This helps expose the model to more aspects of the data distribution, and hence, *generalize* better.

Keras can make this happen by allowing us to define a series of random transformations to be performed during training and evaluation. Gist 3.5 shows how this can be accomplished:

```
>> datagen = ImageDataGenerator(
  rotation_range=40,
  width_shift_range=0.2,
  height_shift_range=0.2,
  shear_range=0.2,
  zoom_range=0.2,
  horizontal_flip=True,
  fill_mode='nearest'
)
```

GIST 3.5: Augmenting the images using an image generator.

These are some of the many transformations provided by *Keras*. Let's briefly look at them:

- **rotation_range**: the range (in degrees) within which to randomly rotate images.
- **width_shift_range**: the range (as a fraction of width) within which to randomly translate images vertically.
- **heighth_shift_range**: the range (as a fraction of height) within which to randomly translate images horizontally.
- **shear_range**: range for randomly shearing transformations.
- **zoom_range**: range for randomly zooming transformations.

- **horizontal_flip**: whether or not to randomly flip inputs horizontally.
- **fill_mode**: the mode according of which points outside the boundaries of the input are filled in.

If we train our model with such a configuration, our model will never see the same image twice. Still, the images will be heavily *intercorrelated*, as they come from a small number of original images. Obviously, we cannot produce new information but only remix the existing one. Therefore, *augmentation* may not be enough to completely get rid of *overfitting*, and as previously mentioned, the only way to fight further is by using some complementary techniques, including but not limited to, **dropout** and **weight decay**.

3.4 Transfer Learning

There are two different approaches to develop a *CV* application. The first one is to train the network from *scratch*, as described earlier in this chapter. A much faster approach is to download a network that someone else built and trained (very likely on a completely different domain and dataset) some time ago and use it as a starting point to build another network that solves another problem. This approach is called *transfer learning*.

Transfer learning is one of the most important techniques in *CV* these days. In general, training a *NN* from scratch requires collecting and labeling a large amount of data. This is not always feasible for various reasons, including but not limited to *time*, *cost*, and *access* to new information. Transfer learning allows us to build a more accurate model, much faster, and with less data.

Over the last few years, several research labs have published their *NN* models trained on massive *GPUs* to achieve state-of-the-art results. All this effort comes to benefit individual contributors as they can simply download these models, including their weights, and use them as a starting point to build new *NN* models. Transfer learning refers to the knowledge transfer from one domain to another through a pre-trained network to solve a different problem.

In the remainder of this section, we explain transfer learning in more detail and outline the reasons why it is important. We also analyze different transfer learning scenarios and how to use them appropriately. Finally, we present examples of utilizing transfer learning to solve real-world problems.

Let's get started.

3.4.1 Importance

As the name suggests, transfer learning is about knowledge transfer from one problem to another through the use of pre-trained models. It's a very hot topic in *DL* at the moment as it enables us to train deep *NNs* with relatively little data and in a relatively short time. The importance of transfer learning comes from the fact that, in most real-world problems, we usually do not have enough labeled images to train such complex models (see figure 3.17).

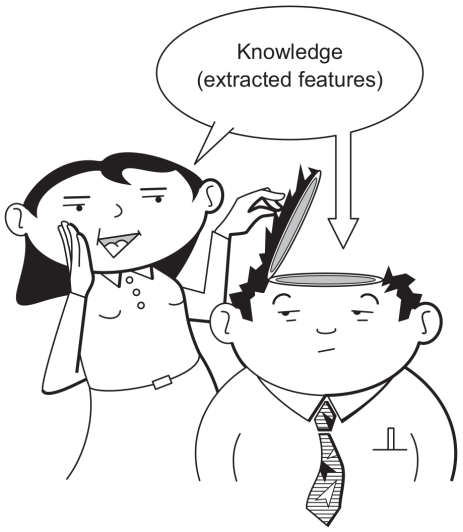

FIGURE 3.17: Knowledge transfer.

The idea is pretty simple. First, we *train* a network on a large dataset. During that process, the network *extracts* a large number of useful features that can be used to detect objects in this dataset. Next, we *transfer* the extracted features to a new network. Finally, we *re-train* that network on a new dataset to solve a new problem. Transfer learning is a great way to shortcut the process of collecting and training a new dataset by using the weights of other models that were built from publicly available datasets. The models can then be downloaded and used directly or embedded into new models for completely different tasks.

Someone may ask why not training the network directly on the new dataset to solve the problem. To answer this question, we first need to understand the main challenges transfer learning is trying to solve.

Deep *NNs* are extremely data-hungry and depend on huge amounts of labeled data to achieve high accuracy. In practice, very few teams train entire *CNNs* from scratch. This happens for two main reasons:

- **Data problem**: Training a NN from scratch requires many data to get decent results. It is very common not to have data of sufficient size to solve a problem. It is also very expensive to acquire and label new data as this usually involves humans.
- **Computation problem**: Even if we have plenty of data, it is computationally expensive to train a NN on massive data as this can take weeks. Also, training a NN is an iterative process as this usually requires experimentation with different hyperparameters.

Another benefit of using *transfer learning* is that of helping the model *generalize* better and avoid *overfitting*. When making *ML* predictions, the model is faced with conditions that it might have never seen before and doesn't know how to deal with. The model is asked to perform well on data not exactly similar to the training one.

For instance, when we deploy a dog classifier, the users use a wide range of cameras, each with its own quality and resolution. Furthermore, images are taken during

various weather conditions. To train a model on all these different nuances, we either have to account for each of these cases by *acquiring* more images or build a more robust model that does better at *generalizing* to new cases. Since it's not practical to account for all possible cases that the model will encounter in the future, *transfer learning* can help us deal with these novel scenarios. It's important for the model to go beyond tasks and domains where labeled data is plentiful. Transferring features extracted from another network that has seen millions of images will make the model more robust. We will fully understand this concept when we explain how *transfer learning* works in the following sections.

3.4.2 Definition

Now that we understand the problems transfer learning is trying to solve let's look at a formal definition. Transfer learning is the transfer of the *feature maps* that the network has learned from one task - with a large amount of data - to a new task - where data is not plentifully available. More specifically, the model is initially trained on a rather generic dataset (e.g., *ImageNet*). Then, one or more of its top layers are replaced, and the model is re-trained on a new dataset not necessarily similar to the original one (e.g., *ImageCLEFmed*).

As we discussed previously, in order to train an image classifier that achieves human-level accuracy, we'll need massive amounts of data, large computational power, and plenty of time. As you can imagine, this would be a big obstacle for individual practitioners and small labs. Luckily, researchers around the globe built state-of-the-art models trained on large datasets, such as *ImageNet*, *COCO*, and *Open Images*, and offered them for free to the general public. That means we should never have to train an image classifier from scratch again unless we have an exceptionally large dataset and massive computational power. But, even if this is the case, we might still want to use transfer learning to fine-tune the pre-trained network on our proprietary dataset.

When we say train the model from *scratch*, we mean that the model starts with *zero* knowledge of the world, and hence the model's structure and parameters begin as random guesses. In other words, the weights of the model are initialized randomly, and hence they need to go through a training process to be *optimized*.

The intuition behind transfer learning is that if a model is trained on a large and generic dataset, it will effectively serve as a good representation of the *visual world*. We can then leverage the *feature maps* it has learned by transferring what it learned to our model and using that as a base starting model for our own task.

Transfer learning starts with a base network, which was trained on a base dataset. Then, its features are *re-purposed* to an entirely new network using an entirely new dataset. This process tends to work exceptionally well as long as the base network features are as *generic* as possible and, therefore, not tight to a specific task.

Let's give a concrete example to better understand how transfer learning is used in practice. Suppose we want to train a model that classifies *dog* and *cat* images. We need to collect hundreds of thousands of images for each class, label them, and train our network from scratch. Another option is to do *knowledge transfer* from a pre-trained network.

First, we need to find a dataset that has similar characteristics to our task at hand. This involves searching for open-source datasets. Let's assume that we choose the *ImageNet*

dataset since we already know that it contains many dog and cat images. So, any network trained on this dataset will be already familiar with both objects. Next, we need to choose the pre-trained network. There are many state-of-the-art architectures noways. Let's assume that we choose the *VGG16* network since we already know that it was trained on the *ImageNet* dataset.

In order to adapt the *VGG16* network to our task, first, we need to download it together with its pre-trained weights. Next, we need to remove the classifier part and add a new one that better fits our task. Finally, we need to re-train the network. This is called *using a pre-trained model as a feature extractor*. We will discuss the different types of transfer learning later in this section.

A pre-trained model is a network that has been previously trained on a large dataset, typically on a large-scale image-classification task. We can either directly use the pre-trained model *as is* to run our predictions, or we can use the pre-trained *feature extraction part* of the network and add a classifier that better fits our task. The classifier could be one or more *densely-connected layers* or even a traditional *ML* algorithm such as *SVM*.

Figure 3.18) illustrates an example of applying transfer learning to the *VGG16* network. As we can see, first, we freeze the feature extraction part of the network and remove the classifier part. Then, we append a new classifier by adding a *softmax layer* with two *hidden units*.

Training the new model will be a lot faster than training the network from scratch. Here, for example, the number of **trainable** parameters is *50K*, whereas the number of **non-trainable** parameters is *14M*. These non-trainable parameters are already trained on a large dataset, and thus, we froze them to use the extracted features in our problem. With this new model, we don't have to train the entire *VGGNet* from scratch because we only have to deal with the newly added *softmax layer*.

Furthermore, we gain much *better performance* with transfer learning because the new model has been trained on millions of images, allowing the network to understand finer details of the object nuances, which ultimately leads to *generalizing better* on new, unseen images.

Now, let's examine how transfer learning works internally.

3.4.3 Internals

So far, we have learned what transfer learning is and what type of problems it is trying to solve. In this section, we will see why transfer learning works, what exactly is being transferred from one task to another, and how a network that is trained on one dataset can perform well on a different, possible unrelated, dataset.

When we're training a *CNN*, the network extracts features from an image in the form of *feature maps*. The feature maps are the outputs of each layer in a *NN* after applying the weights filter. They are representations of the features that exist in the training set. They are called like that because they map where a certain kind of feature is found in the image. *CNNs* look for features such as *straight lines*, *edges*, or even *objects*. Whenever they spot these features, they report them to the feature map. Each feature map is looking for something else (see figure 3.19).

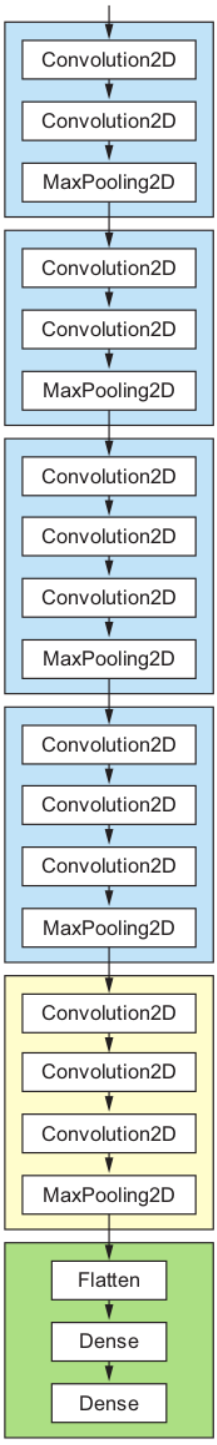

FIGURE 3.18: Transfer learning and fine-tuning using VGG16.

Recall that *NNs* iteratively update their weights during the training sequence of **feed-forward** and **backpropagation**. We say that the network has been trained when we go through a series of training iterations and hyperparameter tunings until the network yields satisfactory results. When training is complete, we export two main items: 1) the *network architecture* and 2) the *trained weights*. So, when we use a pre-trained network, we essentially download both the architecture as well as the weights.

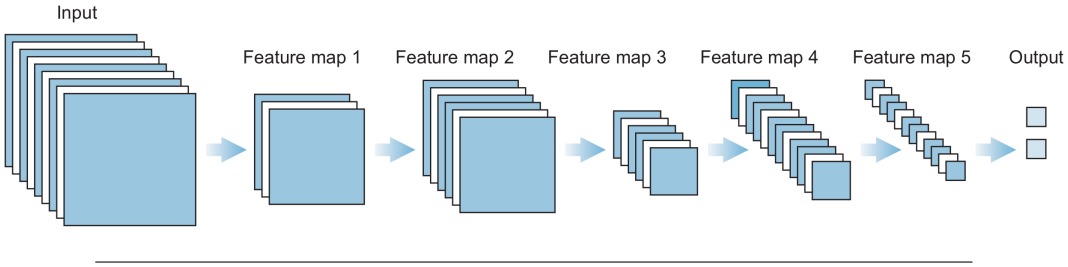

FIGURE 3.19: Stacking and chaining feature maps.

During training, the model learns only the features that exist in the training dataset. But when we download large models trained on a vast amount of data, the accompanying features are now available for us to use. This is extremely important because these pre-trained models have identified features that aren't part of our small dataset, ultimately leading to the build of more a robust model for our task.

In vision tasks, there's so much stuff for *NNs* to learn about the training data. For example, there are **low-level** features like edges, corners, round shapes, curvy shapes, and blobs. Also, there are **mid-level** and **higher-level** features like eyes, circles, squares, and wheels. There are so many details in the images that *CNNs* can pick up on. But if we have only *1K* or even *25K* images in our training data, this may not be enough for our model to learn all those things. By utilizing a pre-trained network that someone else built, we literally embed all this knowledge into our *NN* to give it a big boost, both in terms of performance and speed.

The *NN* learns the features **step-by-step** in an increasing level of complexity **layer-after-layer**. These are called feature maps. The deeper we go through the network layers, the more image-specific features are learned. See figure 3.20 for an illustration.

First, the top layer detects low-level features such as curves and edges. Then, the first layer's output becomes the input to the second layer, which produces higher-level features such as squares and semi-circles. The next layer assembles the output of the previous layer into parts of familiar objects. Finally, a subsequent layer detects the objects. As we go through more layers, the network yields an activation map representing more and more complex features. The deeper we go into the network, the more responsive to a larger region of the pixel space the filters become. Higher-level layers amplify aspects of the received inputs that are important for discrimination and suppress irrelevant variations.

Consider the example of figure 3.20 again. Suppose that we are building a model that detects human faces. We notice that the network learns low-level features like lines, edges, and blobs in the first layer. These low-level features appear not to be specific to a particular dataset or task but rather generic. The mid-level layers then assemble those lines together to recognize shapes, corners, and circles. We notice that the extracted features start getting a little more specific to our task. So we see that the mid-level features contain combinations of shapes that form objects in the human face like eyes and nose. As we go deeper through the network, we notice that the features transition from generic to specific, and, by the last layer of the network, they form high-level features that are very specific to our task. So we start seeing parts of human faces that distinguish one person from another.

Let's take this concept and compare the feature maps extracted from four different models that were trained to classify *faces*, *cars*, *elephants*, and *chairs* (see figure 3.21).

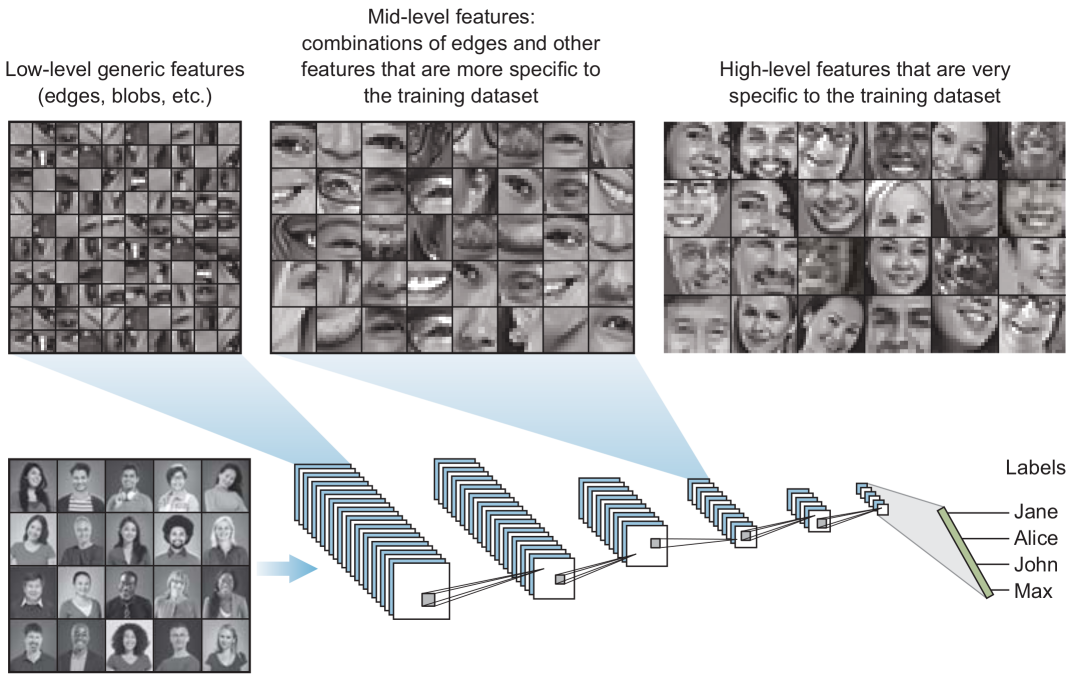

FIGURE 3.20: Complexity of feature maps increases as we go deeper.

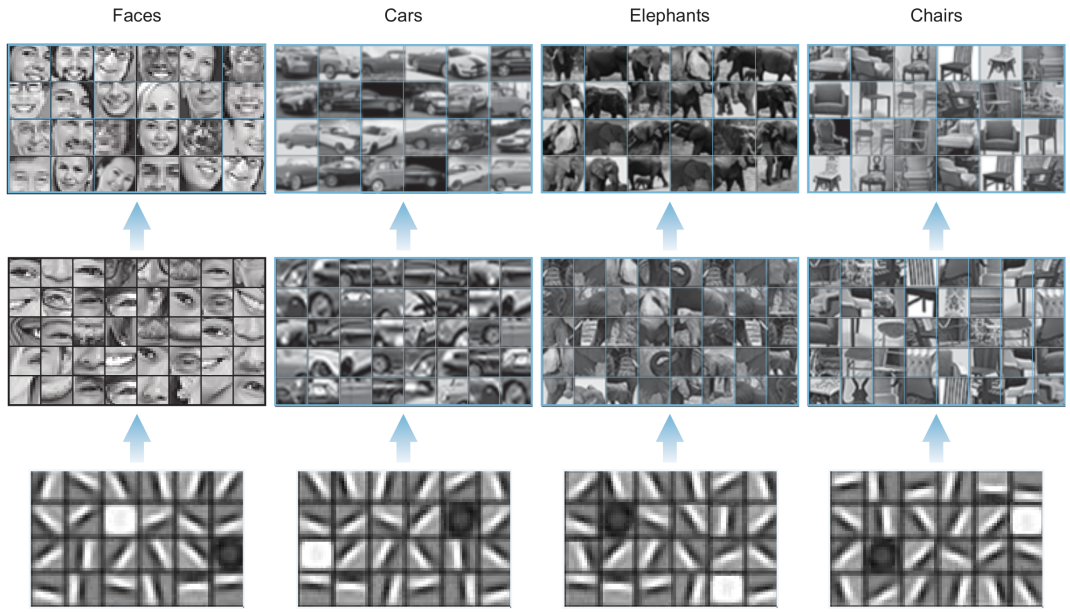

FIGURE 3.21: Comparison of feature maps extracted from four models.

Notice that the features of the earlier layers are very similar for all models. They represent low-level features like edges, lines, and blobs. This means that models that are trained on one task capture similar relations in the data type in the earlier layers of the network and can be easily re-used for different problems in other domains. The deeper we go into the network, the more specific the features become, until the network overfits its training data, and eventually becomes harder to generalize to different tasks. The lower level features are almost always transferable from one task to another because they contain generic information like the structure and the nature of

how images look. Transferring information like lines, dots, curves, and small parts of the objects is very valuable for the network to learn faster and with less data on the new task.

The transferability of features that are extracted at later layers depends on the similarity between the original and the new datasets. The idea here is that all images have shapes and edges, so the early layers are usually transferable between different domains. We can only identify differences between objects when we begin to identify higher-level features such as eyes in faces or wheels in vehicles. Only then can we tell that this is a face because it has eyes or a vehicle because it has wheels. Based on the similarity of the source and target domains, we can decide whether to transfer only the low-level features or all the high-level features or somewhere in between. This is driven by the observation that the later layers of the network become progressively more specific to the details of the classes contained in the original dataset.

3.4.4 Approaches

There are three main transfer learning approaches, as follows:

- Using pre-trained network as a classifier.
- Using pre-trained network as a feature extractor.
- Using pre-trained network and fine-tuning it.

Any of these approaches can be effective and efficient toward the development and training of deep *CNNs*. It may not be evident from the beginning which of them will yield the best results for our task; therefore, some experimentation may be required here. This section explains each of these approaches in more detail and gives examples of how to proceed with the implementation.

Using pre-trained network as a classifier: The pre-trained model is used directly to classify new images without making any changes to it or performing any extra training. We just need to download the network along with its weights and run the predictions directly on our new data. This approach works well when the new domain is very similar to the one the pre-trained network was trained on. See figure 3.22) for an illustration.

In the dog breed classification task, we can simply use the *VGG16* network trained on the *ImageNet* data to run predictions directly. Because *ImageNet* already contains many dog images, a significant portion of the pre-trained network's representational power may be dedicated to features specific to differentiating between dog breeds.

It worth pointing out that using a pre-trained network as a classifier doesn't really involve any layer freezing or extra training. Instead, we simply take a network that was previously trained on a similar problem and start using it directly on the new task.

Using pre-trained network as a feature extractor: Here, we take the *VGG16* pre-trained network, freeze the weights of the first 13 convolutional layers, and replace the old classifier with a new densely-connected layer that will be trained from scratch. See figure 3.23 for an illustration.

This approach works well when the new task is similar to the task the pre-trained network was trained on. Since the *ImageNet* dataset has many dog and cat images,

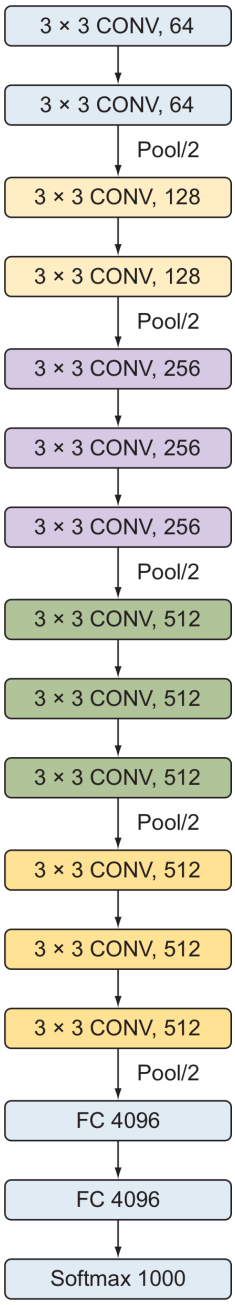

FIGURE 3.22: Using pre-trained network as a classifier.

the feature maps that the network has learned contain many dog and cat features that are applicable to the new task. This means that we can utilize the high-level features extracted from the *ImageNet* dataset into this new task.

This can be achieved by *freezing* all the pre-trained network layers and *training* only the classifier part that we just added. In other words, we only add a new classifier, which will be trained from scratch, on top of the pre-trained model, to re-purpose the feature maps learned previously for the new dataset.

We remove the classifier part because it's close related to the original classification task, and consequently, to the set of classes the model was trained on. For example,

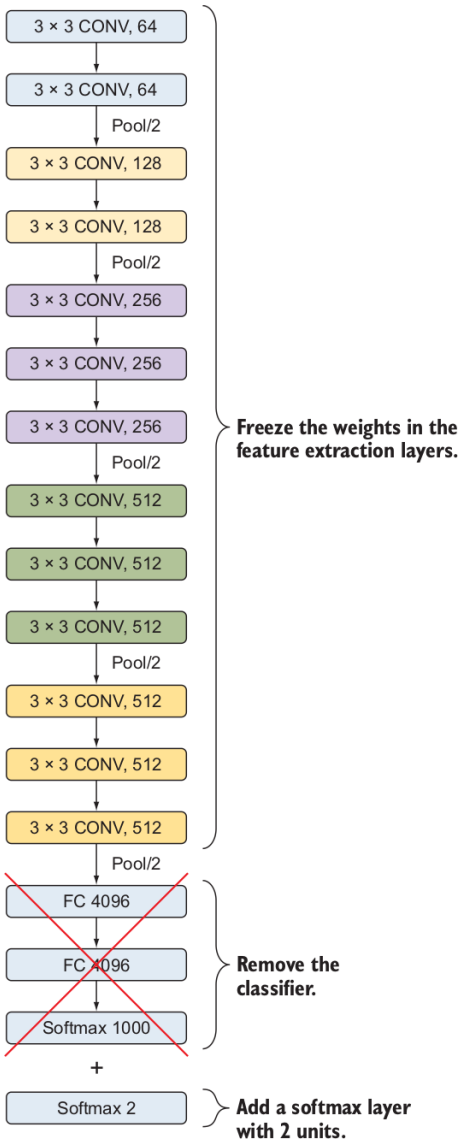

FIGURE 3.23: Using pre-trained network as a feature extractor.

ImageNet has *1K* classes, and hence, the classifier has been trained to overfit the training data into these *1K* classes only. However, in the dog vs. cat task, we only have two classes. So it's a lot more effective and efficient to train a new classifier from scratch to overfit these two classes only.

Using pre-trained network and fine-tuning it: The first two approaches are useful when the target domain is relatively *similar* to the source domain. However, what if this is not the case? Can we still take advantage of transfer learning? The answer is, of course, yes! Transfer learning works excellent even when the source and target domains are entirely *different*. We simply need to extract the right feature maps from the source domain and *optimize* them to fit the target domain.

A better definition of fine-tuning is *freezing* some of the network layers used for feature extraction and simultaneously *training* both the non-frozen and the newly added classifier layers of the pre-trained model. In other words, while we *re-train* the feature extraction layers, we also *fine-tune* the higher-order feature representations to make

them more *relevant* for the new task. See figure 3.24 for an illustration.

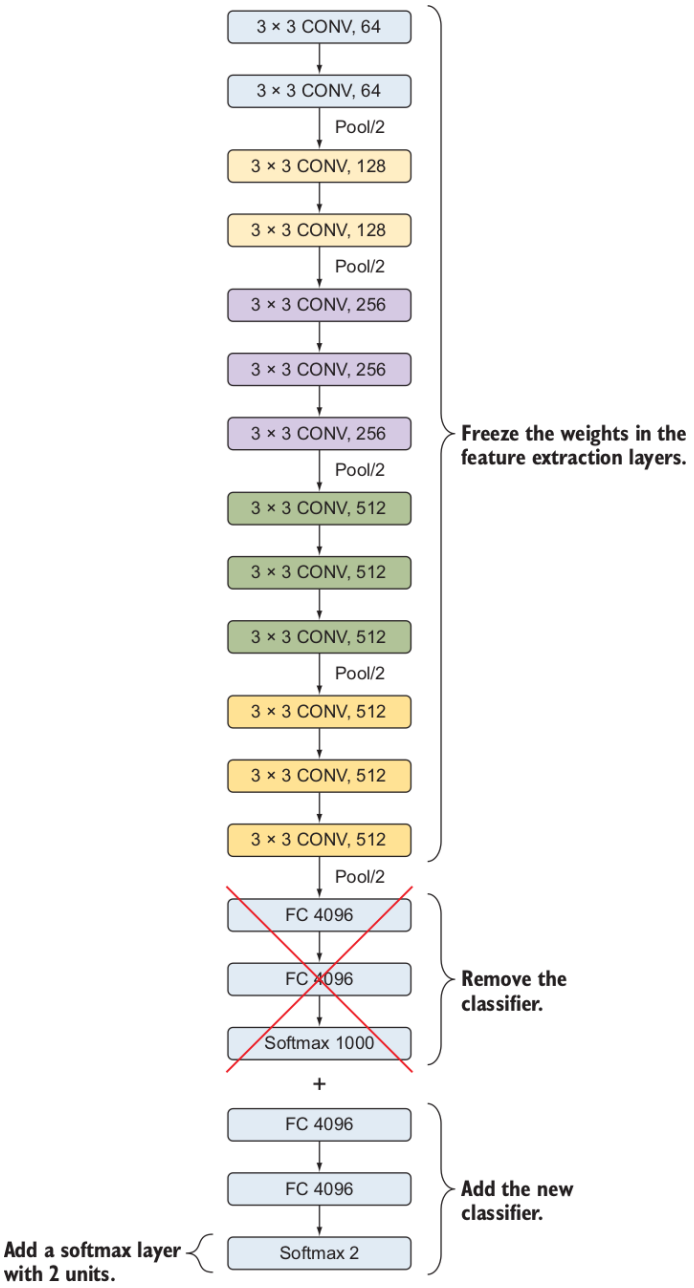

FIGURE 3.24: Using pre-trained network but fine-tuning it.

3.4.5 Conclusion

Figure 3.25 illustrates the different knowledge transfer approaches from pre-trained networks. If we download the entire pre-trained network and just run predictions on it, then we use it as a *classifier*. If we freeze the convolutional layers only, then we use it as a *feature extractor*. Finally, if we freeze a few layers and jointly train both non-frozen and newly-added layers, then we *fine-tunning* it. The big question now is: up to which feature map should we freeze the network? To answer this question, let's first recall how fine-tuning works.

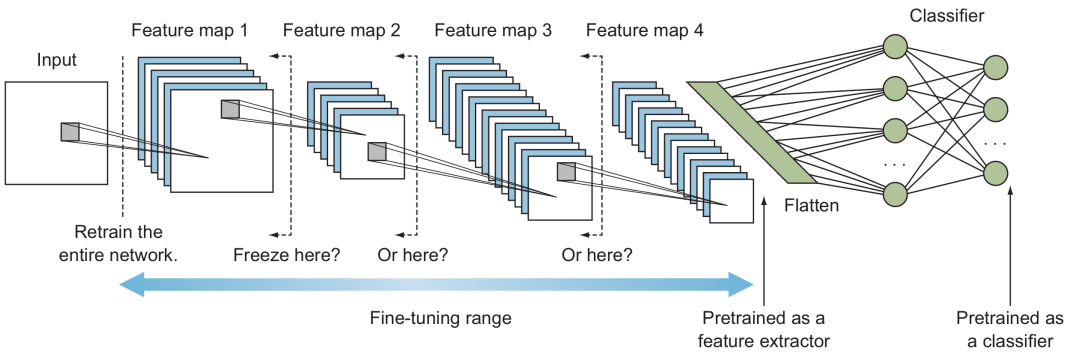

FIGURE 3.25: Fine-tuning approaches.

As we discussed earlier, feature maps that are extracted early in the network are *generic*; however, as we go deeper into the network, feature maps are getting more *specific*. So, depending on the similarity between the *source* and the *target* domains, we may decide to freeze the network earlier rather than later (or vise versa). For example:

- If the domains are very similar, we might decide to freeze up to the 4^{th} feature map.
- If the domains are very different, we might decide to freeze up to the 1^{st} feature map, then re-train all remaining layers.

Between these two options, there is a wide range of options that we can apply. Finding the right fine-tuning level is *more an art than a science*; however, there are some guidelines to help us make a choice more intuitively. The decision is a function of two factors: 1) the *amount of data* that we have and 2) the *level of similarity* between the source and the target domain. We will explain these two factors in more detail in a bit; however, before that, let's discuss why fine-tuning is always better than training from scrats.

When we train a network from scratch, we normally initialize its weights randomly. As a result, there is no guarantee that they will get values close to the optimal ones. And if they are far away from the optimal ones, the optimizer will take a long time to converge. This is when fine-tuning can prove extremely useful. The weights of the pre-trained network have already been optimized based on some general-purpose data. Thus, when we use this network in our problem, we start with the weights it ended with. This makes the network converge much faster than if it had to randomly initialize the weights. So, even if we decide to re-train the entire pre-trained network, starting with the optimized weights will converge faster than training the network from scratch with randomly initialized weights.

Recall that early convolutional layers extract generic features, then they get more specific to the training data the deeper we go through the network. With that said, a granular level of detail from a pre-trained model can be chosen for feature extraction. For example, if the new task is entirely different from the one that the model was trained originally (e.g., ImageNet vs. ImageCLEFmed), then, perhaps, the output of the pre-trained model after the first few layers would suffice. But if the new task is quite similar to the original one, then the output from layers much deeper in the model can be used.

Now, let's go back to the important factors. As we mentioned before, choosing the appropriate level of transfer learning is a function of *two* important factors:

- **Dataset size**: When the target dataset is small, the network will tend to overfit the new data. In this case, we will most likely want to do less fine-tuning and rely more on the source dataset.
- **Domain similarity**: If we want to classify cars vs. boats, ImageNet will suffice as it contains many such images. However, if we want to classify breast cancer images, this is an entirely different domain, and thus, it will most likely require a lot of fine-tuning.

These *two* factors result in the following *four* scenarios:

- Target is small and similar to the source.
- Target is large and similar to the source.
- Target is small and different from the source.
- Target is large and different from the source.

Let's discuss two of these scenarios a bit more to understand their main characteristics for navigating our options.

Target is small and different from the source: Since the data is different, it might not be best to freeze the higher-level features of the pre-trained network, because they contain more data-specific features. Instead, it would work better to re-train layers from somewhere earlier in the network. However, since our data is small, fine-tuning the entire network might not be a good idea, because doing so will make it prone to overfitting. A mid-way solution would work better in this case, e.g., freeze approximately the first third or half of the pre-trained network. After all, the early layers contain very generic feature maps that would be useful for our data, even if it's very different.

Target is large and different from the source: Since the new data is large, we might be tempted to just train the entire network from scratch and not use transfer learning at all. However, in practice, it's often still very beneficial to initialize weights from a pre-trained model, as we discussed earlier. Doing so makes the model converge faster. In this case, data is large, and thus, it provides us with the confidence to fine-tune through the entire network without having to worry about overfitting.

Previously in this section, we learned about the two main factors that help us determine which transfer learning approach to use, i.e., the *size* of the target dataset and the *similarity* between the source and the target domains. These *two* factors result in *four* scenarios as described in table 3.1.

Target Data	Domain Difference	Transfer Learning Approach
Small	Very Similar	Use pre-trained network as feature extractor
Large	Very Similar	Fine-tune last one third of the network
Small	Very Different	Fine-tune last two thirds of the network
Large	Very Different	Fine-tune through the entire network

TABLE 3.1: Fine-tuning summary.

Finally, figure 3.26 illustrates the guidelines for the appropriate fine-tuning level to use in each of these *four* scenarios.

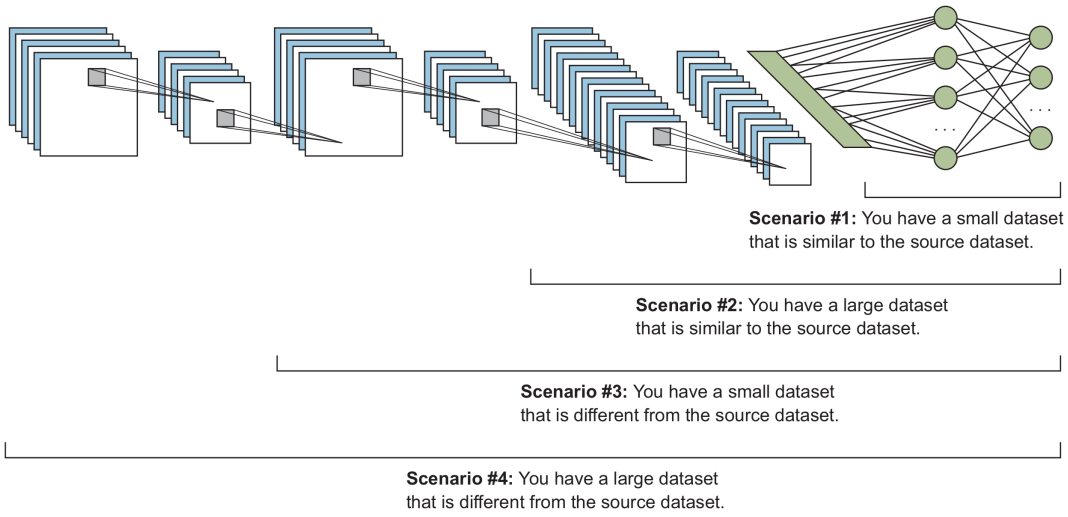

FIGURE 3.26: Fine-tuning guidelines.

Chapter 4

ImageCLEFmed Concept Detection Task

4.1 Task Description

The first step towards the understanding of medical images, such as *X-rays*, *CTs*, *MRIs*, *Ultrasounds*, *PETs*, etc., is to identify potential abnormalities on them. Clearly, this is the job of *multi-label* classification where, given a medical *image*, we identify all relevant medical *terms* in it (see figure 4.1). These terms represent keywords from a large and continuously growing corpus of medical terms.

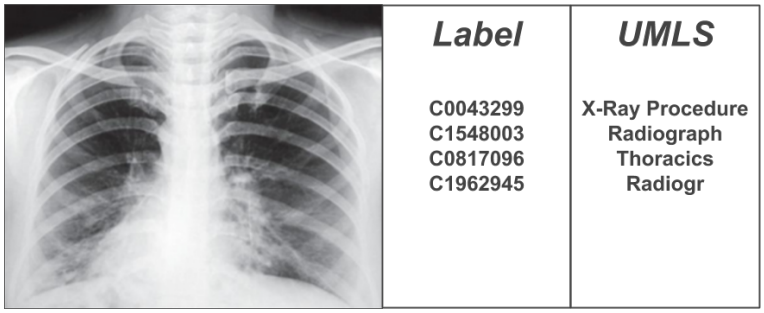

FIGURE 4.1: Sample image and labels from ImageCLEFmed 2019.

In this chapter, we address the problem of *multi-label* classification to decide the presence (or absence) of medical *terms* in medical *images*. We do so through a *non-official* participation in the last *four* editions of the *Concept Detection* task within the *ImageCLEFmed* campaign (Eickhoff et al., 2017; Herrera et al., 2018; Pelka et al., 2019). That means that, although our system *outperformed* all previous winners, our participation cannot be considered valid since it occurred after the *ground truth* datasets had been released to the general public. We can think of this effort as some sort of *ex-post* analysis task.

Having that in mind, in the following sections, we will talk about the main aspects of our system, and we will present how it performed on the numerous *configurations* we tried out along the way. Let's get started!

4.2 Exploratory Analysis

There are a few datasets available nowadays to play around with if we are into medical imaging and multi-label classification. Let's briefly talk about the four most recent ones from the *ImageCLEFmed* campaign. According to tables 4.1 and 4.2, it's pretty obvious that as we go back in time the datasets become more *diverse* and hence more *noisy*. That means we deal with a bigger number of images and a wider range of concepts. As a result, as we go back, we expect it will be harder for a *DL* algorithm to build an accurate model.

If we look carefully at the winners of each year, it will become evident that our initial assumption was indeed valid 4.3. In other words, as we go from 2017 to 2020, the winning teams managed to build more *robust* models, primarily because the dataset was less noisy compared to the ones from the previous years, and secondarily because of the recent advancements in *DL*.

year	dataset	column	total	unique
2017	train	concept	27533	20812
2017	train	images	976504	167748
2017	test	concept	6453	6453
2017	test	images	54133	9607
2018	train	concept	111156	111156
2018	train	images	6652982	222305
2018	test	concept	35878	35878
2018	test	images	323012	9938
2019	train	concept	8449	5528
2019	train	images	432753	70786
2019	test	concept	2456	2456
2019	test	images	64071	10000
2020	train	concept	6094	3047
2020	train	images	907718	80723
2020	test	concept		
2020	test	images		

TABLE 4.1: Simple statistics on the number of images per concept.

The other thing worth mentioning here is illustrated in figures 4.2 and 4.3. What these two figures show is that *not* all labels were born equal! That means that some labels are associated with way too *many* images whereas others are extremely *rare*; i.e., they are associated with one or two images at most. This property is important because it allows us to achieve the same accuracy as if we had considered *all* labels whatsoever by choosing only a *subset* of them. In other words, by doing so, we can significantly *speed-up* the training process and eliminate the *noise* while maintaining most of the original information. In fact, this is exactly what we did in our experiments; hence, we had the opportunity to prove it firsthand.

year	dataset	column	total	unique
2017	train	image	167748	167748
2017	train	concepts	976504	20812
2017	test	image	9607	9607
2017	test	concepts	54133	6453
2018	train	image	222305	222305
2018	train	concepts	6652982	111156
2018	test	image	9938	9938
2018	test	concepts	323012	35878
2019	train	image	70786	70786
2019	train	concepts	432753	5528
2019	test	image	10000	10000
2019	test	concepts	64071	2456
2020	train	image	80723	80723
2020	train	concepts	907718	3047
2020	test	image		
2020	test	concepts		

TABLE 4.2: Simple statistics on the number of concepts per image.

year	team	score
2017	Aegean AI Lab	0.1583
2018	UA PT Bioinformatics	0.1108
2019	AUEB NLP Group	0.2823
2020	AUEB NLP Group	0.3940

TABLE 4.3: Winning teams from previous campaigns.

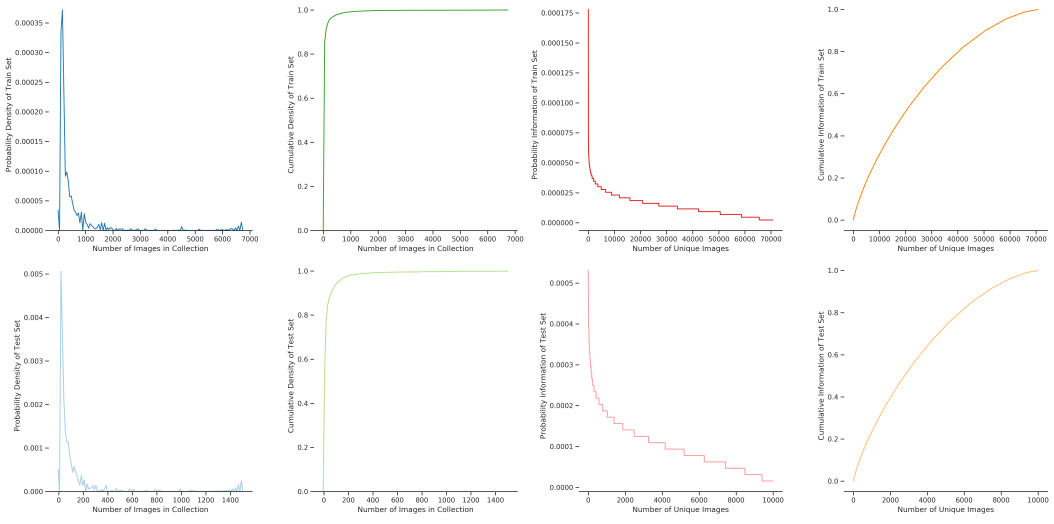

FIGURE 4.2: Distribution of images from ImageCLEFmed 2019.

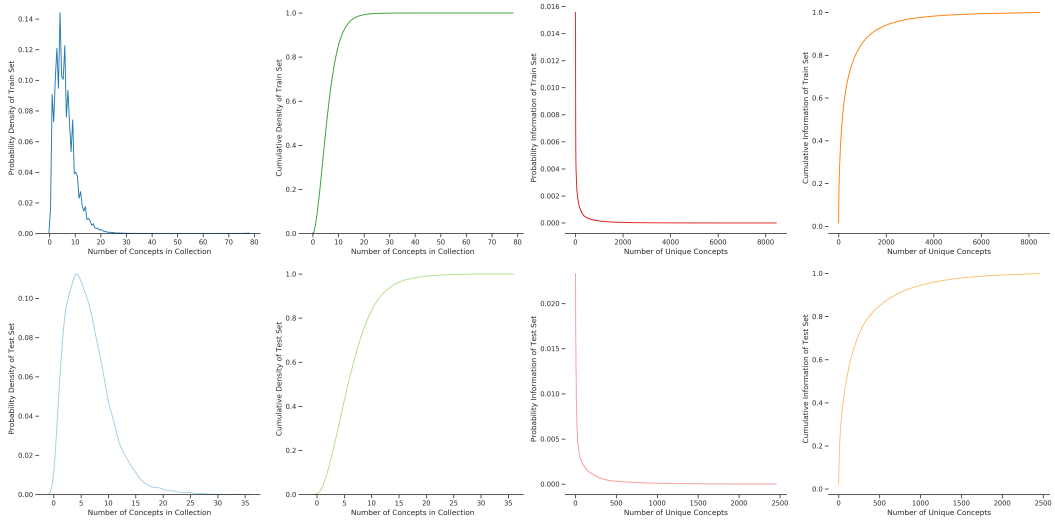

FIGURE 4.3: Distribution of concepts from ImageCLEFmed 2019.

4.3 Related Work

DL techniques are broadly used for image classification tasks in the healthcare sector. Most often, pre-trained *CNNs* on the *ImageNet* image database (Deng et al., 2009) are used as **image encoders** followed by a *FFNN* that serves as **image classifier** (Esteva et al., 2017; Rajpurkar et al., 2017; Rajpurkar et al., 2018). However, *ImageNet* consists of just stock images from everyday life, and thus it differs significantly from a very specialized repository of medical images. As a result, a common practice among scientists is to *fine-tune* the *pre-trained* model to achieve better results. For example, (Esteva et al., 2017) fine-tuned the *InceptionV3* model to classify skin lesions into malignant or benign, scoring results close to the ones predicted by dermatologists. This breakthough revealed that pre-trained models could indeed be used in medical imaging tasks when fine-tuned, despite the *chaotic* differences between *general*-purpose and *special*-purpose images.

Similarly, *CheXNet* (Rajpurkar et al., 2017) followed a *DL* approach to classify *X-rays* of the *ChestXray14* dataset (Wang et al., 2017) to fourteen labels of thoracic diseases. Also, (Rajpurkar et al., 2017) used the *DenseNet-121* pre-trained model (Huang et al., 2017) to encode images, adding a *FFNN* to assign one or more of the fourteen classes to each image. The authors evaluated the predicted results with the F_1 score and reported state-of-the-art results. In a later work, (Rajpurkar et al., 2018) presented *CheXNeXt*, which consisted of an ensemble of ten networks with the same architecture as *CheXNet*. First, an ensemble from multiple *CheXNet* networks was used to re-label the *ChestX-ray* dataset to correct its false labels. Then, the networks were re-trained, now on the re-labeled data, and an ensemble of the ten best was used for the final predictions.

As we discussed already, the *Concept Detection* tasks of the *ImageCLEFmed* campaign aim to detect abnormalities in medical images. Most of the participated teams in recent years (Eickhoff et al., 2017; Herrera et al., 2018; Pelka et al., 2019) employed some type of *DL* system. In 2017, the top system from the same lab (Katsios and Kavallieratou, 2017) outperformed all the other methods. In 2018, the winning team (Pinho and Costa, 2018) used an **Adversarial Autoencoder** for unsupervised feature learning,

whereas the second team (Wang et al., 2018) used a multi-label classification method based on the *InceptionV3* pre-trained model. In 2019, the winning system (Kougia, Pavlopoulos, and Androutsopoulos, 2019) also followed a *DL* approach.

4.4 Proposed System

The proposed system is based on the work of (Katsios and Kavallieratou, 2017), yet with many improvements in various areas. Figure 4.4 illustrates the *VGG16* pre-trained model after adding a *FFNN* at the end.

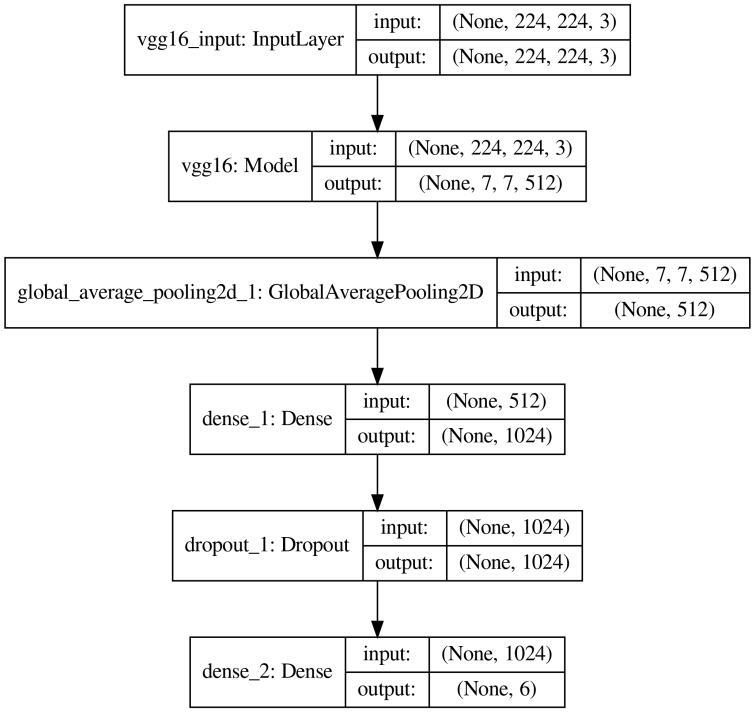

FIGURE 4.4: VGG16 model with appended FFNN.

Other than that, the proposed system follows the standard workflow of *ML* projects (see figure 4.5).

Now, let's briefly discuss the improvements of our system over the one that won the 2017 edition of the *ImageCLEFmed* campaign. Firstly, instead of using all labels, we only used the *top-N*, where *N* is chosen so that the 90% of the total information (i.e. the number of image-label pairs) is considered during training. Secondly, instead of a single pre-trained model and a fixed set of hyper-parameters, we applied several models (e.g., *Xception* and *InceptionV3*, etc.) and hyper-parameters (e.g., *loss functions*, *optimizers*, *learning rates*, *batch sizes*, *epochs*, etc.) through an exhaustive grid-search. Needless to say that the execution time of our experiments increased exponentially as a result of the exhaustive grid-search.

However, the single most important improvement was the selection of an alternative *loss function*. So, instead of the classic *Binary Crossentropy* loss, we choose the *Sigmoid Focal Cross Entropy* loss, which is more appropriate when dealing with extremely imbalanced classes.

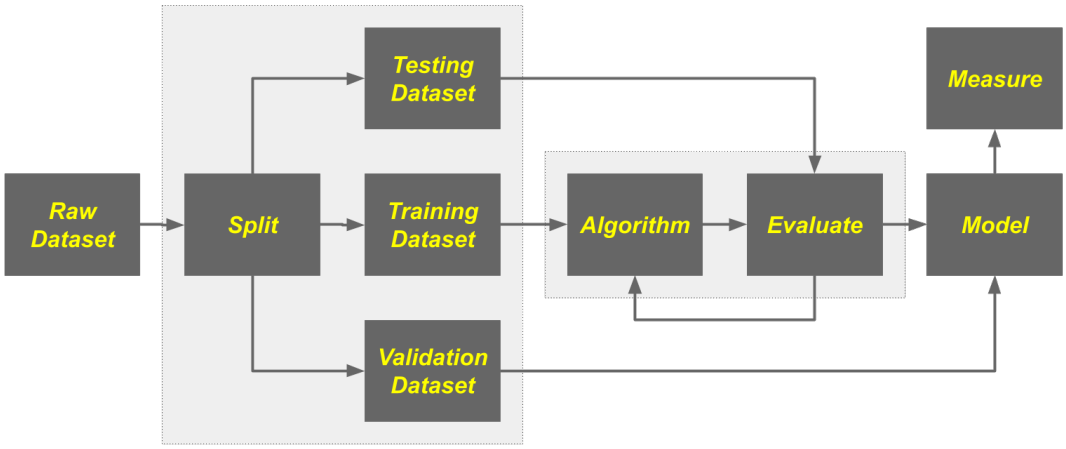

FIGURE 4.5: Workflow of a machine learning project.

Moreover, we *down- or up-scaled* the training images according to the maximum allowed size of the corresponding pre-trained model, and, last but not least, we performed *image augmentation* on the training data according to gist 4.1.

```
>> from keras.preprocessing.image import ImageDataGenerator

>> generator = ImageDataGenerator(
  rescale=1. / 255,
  rotation_range=40,
  width_shift_range=0.2,
  height_shift_range=0.2,
  shear_range=0.2,
  zoom_range=0.2,
  horizontal_flip=True,
  fill_mode="nearest",
)
```

GIST 4.1: Image augmentation logic.

Figure 4.6 lists four *random* images from the 2019 dataset, whereas figure 4.7 lists four *augmented* images of the same reference image, again, from the same dataset.

4.5 Experimental Results

The evaluation of the different models was conducted, first by computing the F_1 score on each image of the testing set (see figure 4.8), and then by calculating the average value.

As already mentioned before, the *testing set*, a.k.a. *ground truth*, was already available to us from the begining. For the *validation set*, we randomly selected 20% of the training images. Recall that the latter is being used for hyper-parameter *tuning* and early *stopping* whereas the former to evaluate how good our model *generalizes* to new, unknown data.

Table 4.4 shows the F_1 scores achieved by our models on all four datasets. Clearly, the proposed system *outperformed* all past winners by a small margin. Lastly, for completeness sake, figures 4.9 and 4.10 illustrate the history and performance of a single

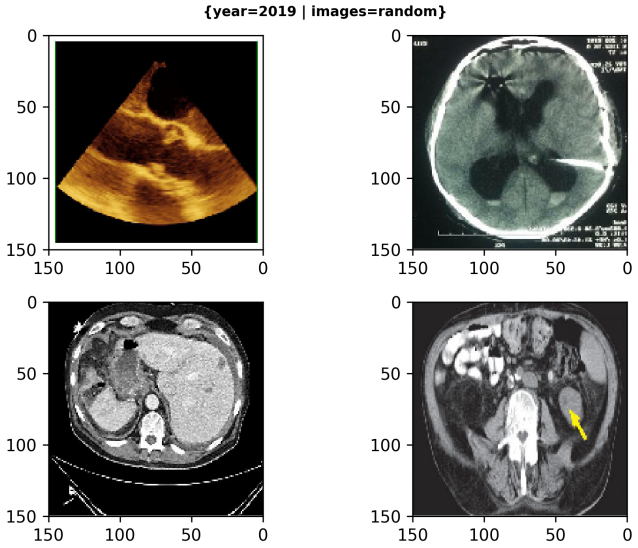

FIGURE 4.6: Random images from ImageCLEFmed 2019.

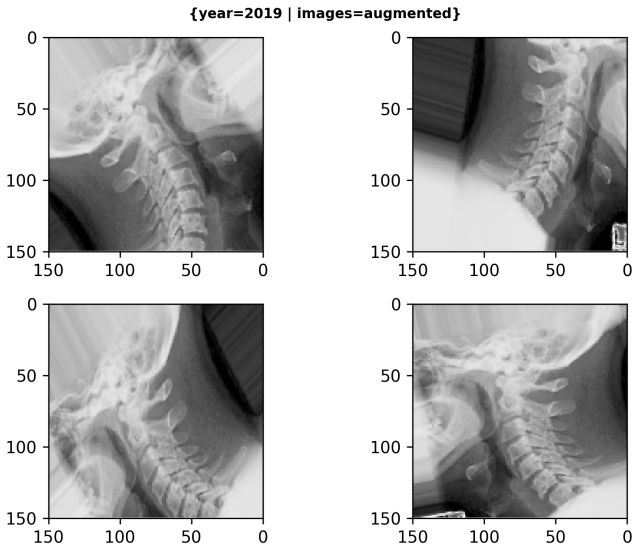

FIGURE 4.7: Augmented images from ImageCLEFmed 2019.

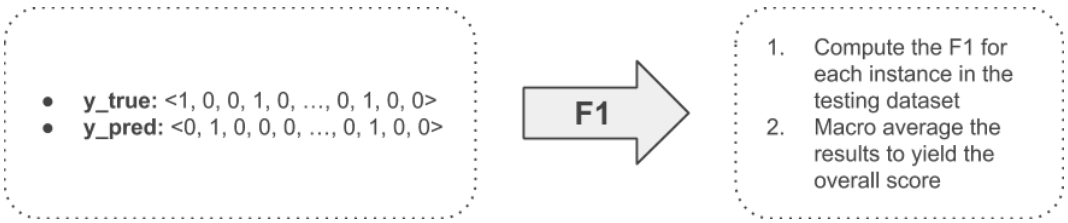

FIGURE 4.8: Model evaluation with F1 score.

experiment (out of many conducted as part of our study) during training, validation, and testing phases.

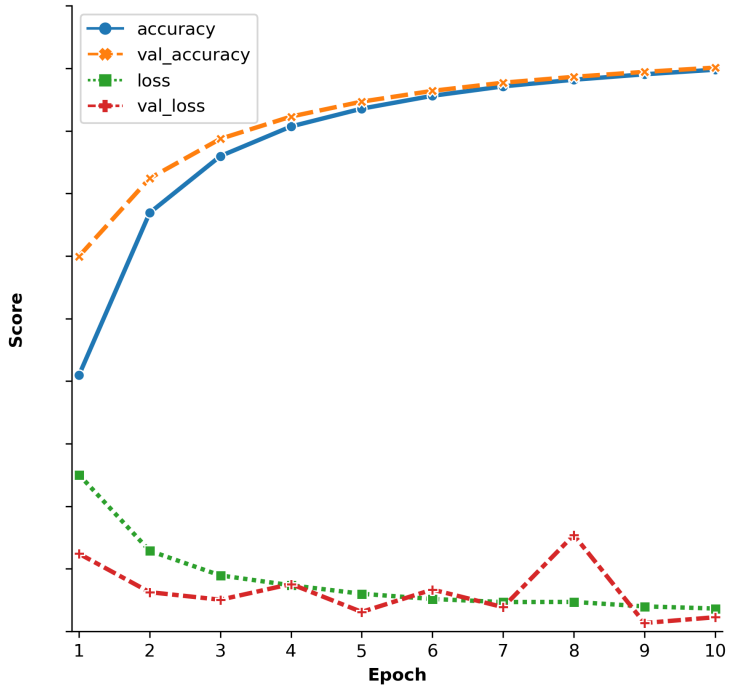

FIGURE 4.9: History of batch=16 | year=2019 | model=VGG16.

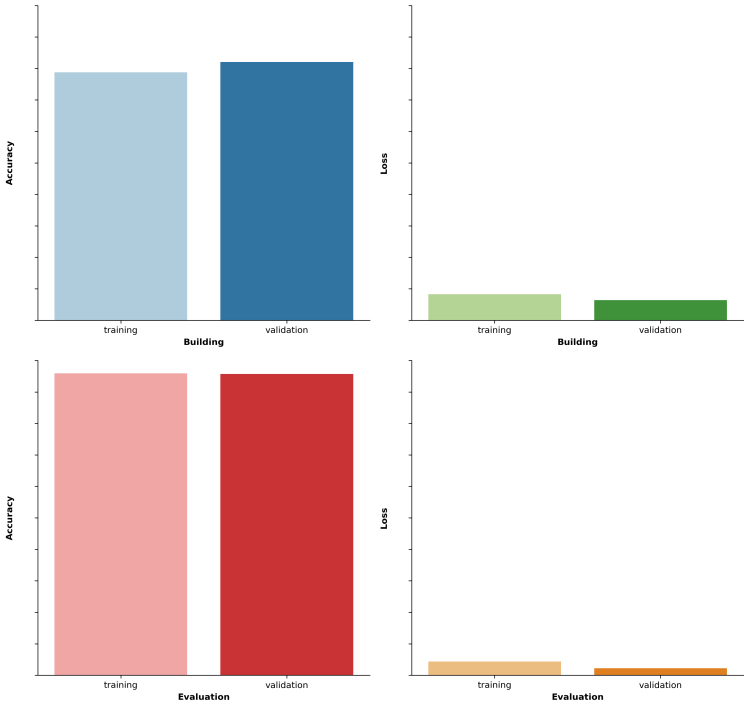

FIGURE 4.10: Performance of batch=16 | year=2019 | model=VGG16.

batch	model	score	year
16	Xception	0.1521	2017
16	VGG16	0.1592	2017
16	ResNet50	0.0569	2017
16	InceptionV3	0.1351	2017
32	Xception	0.1488	2017
32	VGG16	0.1600	2017
32	ResNet50	0.0000	2017
32	InceptionV3	0.1384	2017
64	Xception	0.1521	2017
64	VGG16	0.1572	2017
64	ResNet50	0.0439	2017
64	InceptionV3	0.1447	2017
16	Xception	0.1076	2018
16	VGG16	0.1126	2018
16	ResNet50	0.0402	2018
16	InceptionV3	0.0956	2018
32	Xception	0.1053	2018
32	VGG16	0.1132	2018
32	ResNet50	0.0000	2018
32	InceptionV3	0.0979	2018
64	Xception	0.1076	2018
64	VGG16	0.1112	2018
64	ResNet50	0.0311	2018
64	InceptionV3	0.1024	2018
16	Xception	0.2772	2019
16	VGG16	0.2902	2019
16	ResNet50	0.1037	2019
16	InceptionV3	0.2464	2019
32	Xception	0.2712	2019
32	VGG16	0.2916	2019
32	ResNet50	0.0000	2019
32	InceptionV3	0.2523	2019
64	Xception	0.2772	2019
64	VGG16	0.2866	2019
64	ResNet50	0.0801	2019
64	InceptionV3	0.2639	2019

TABLE 4.4: Performance summary on the test sets.

Chapter 5

Conclusion

5.1 Wrap Up

In this thesis, we addressed the *Concept Detection* task, which aims to assist physicians during the examination of medical images, such as *X-rays*, *CTs*, *MRIs*, *Ultrasounds*, *PETs*, etc. A variety of pre-trained models, hyper-parameter values, and multimodal datasets were involved. More specifically, the datasets were downloaded from the last four years of the *ImageCLEFmed* campaign. The models were pre-trained based on the *ImageNet* image database and fetched through the *Keras API*. Lastly, the hyper-parameter grid-search consisted of *loss functions*, *optimizers*, *learning rates*, *batch sizes*, *epochs*, etc. The proposed system achieved marginally *higher* performance than the winning team in all four campaigns, based on the corresponding *ground truth* datasets.

5.2 Up Next

In future work, we will put more effort into understanding, cleaning, normalizing, and preparing the input data. We also plan to test more complex pre-trained models with a larger number of parameters. Moreover, we will add new hyper-parameters to the grid in addition to optimizing the range of their values. Last but not least, we will experiment with more sophisticated loss functions and optimizers for *multi-label* datasets, which are more robust against *extremely-sparsed* labels and *highly-imbalanced* classes.

Appendix A

Setup Working Environment

Before we start developing a *DL* application, it's required that we setup a proper working environment. Moreover, it's strongly recommended, although not strictly necessary, our application takes advantage of a fast *GPU*. Applications, such as image processing with *CNNs*, are extremely slow on *CPUs*, even fast ones with multiple cores and threads. But even for applications that can realistically run on *CPUs*, we can see a significant speed-up by the use of a modern *GPU*. Thankfully, even if we don't own a workstation or we can't install a *GPU* on it, we can run our code on a leased *AWS EC2* instance. Keep in mind, however, this option can become extremely expensive over time in case of extensive use.

Regardless of whether we will choose to run it locally or remotely, it's strongly recommended that we use a *Linux* distribution such as *Ubuntu 18.04*. Moreover, we will need to install *Python 3.7* programming language - if not already installed - as well as *Keras 2.2.5* and *TensorFlow 2.1.1* packages. If not familiar with open source *DL* tools yet, *Keras* is *TensorFlow*'s high-level *API* for building and training *DL* models. It's an ideal tool for fast prototyping, state-of-the-art research, and at the same time, production deployments.

Firstly, we can use two different modes to develop a *DL* application; **standalone** or **interactive**. Both of them are described below.

For **standalone** development, most people use an *IDE*, such as *PyCharm*. This advanced tool can also assist developers during debugging and troubleshooting. On the other hand, for **interactive** development, the common practice is to use *Jupyter*. *Jupyter* is widely used in the data science community. It's a great way to experiment, do research, and share what we're working on. *Jupyter* stores its source code in a *JSON*-like file, which can be edited easily in a browser. It combines the ability to run *Python* code with text-rich capabilities for annotating what is being done. It also allows data scientists to break long experiments into smaller chunks that can be executed independently. This type of development is called **interactive**, which essentially means we don't have to re-run all our code from the beginning in case something goes wrong at a later stage in our experiment.

Secondly, there are two different ways to execute a *DL* application; **local** (i.e., in a *workstation*) and **remote** (i.e., on the *Cloud*). Both of them are described below.

If we don't own a high-end *GPU* already, running *DL* experiments on the *Cloud* is a dead-simple, low-cost way to get started without having to buy any expensive device. But if we are heavy *DL* users, this setup isn't sustainable in the long run (for more than a month, say). That's because *GPU* instances on *EC2* can become very, very expensive under extensive use. Meanwhile, a second-hand, professional-grade *GPU*, such as

Nvidia Tesla K80, costs just above $500 on eBay (as of May 2020) with an original price tag of $5,000. In other words, *Cloud* is a great way to get start, however, in the long run, a dedicated *workstation* will turn out to be more affordable.

In the following two sections, we will show how to prepare our environment (*PyCharm* vs. *Jupyter*) as well as how to execute our application (*workstation* vs. *Cloud*).

A.1 Setup Local Workstation

The process of setting up a **local** workstation for application development and model training is fairly easy and consists of the following steps:

- Verify OS type.
- Verify Python version.
- System update and upgrade.
- Install BLAS interface.
- Install auxiliary Python libraries.
- Create Python environment and install packages.
- Install and start the PyCharm IDE.
- Install and start the Jupyter Server.

Before we start, we presume that our *OS* is based on *Ubuntu 18.04*, as well as that *Python 3.7* is available on the host machine. If unsure, we can easily verify that by running gists A.1 and A.2.

```
#!/bin/bash
$ uname -a
```

GIST A.1: Verify OS type.

```
#!/bin/bash
$ python3.7 --version
```

GIST A.2: Verify Python version.

The next step is to update as well as upgrade our system using gist A.3.

```
#!/bin/bash
$ sudo apt-get update
$ sudo apt-get upgrade
```

GIST A.3: System update and upgrade.

It's also essential that we install the *BLAS* interface. This will speed-up tensor operations on *CPUs*. See gist A.4 on how to accomplish that.

```
#!/bin/bash
$ sudo apt-get install \
    build-essential cmake pkg-config \
    libopenblas-dev liblapack-dev
```

GIST A.4: Install BLAS interface.

Another requirement is to install a few auxiliary *Python* libraries, including *HDF5*, which will allow us to save large *NN* files to disk, and *Graphviz*, which will enable us to visualize *NN* architectures. We can do so by running gist A.5 from our terminal.

```
#!/bin/bash
$ sudo apt-get install \
    python-pip python-dev python-virtualenv \
    libhdf5-serial-dev python-h5py \
    graphviz
```

GIST A.5: Install auxiliary Python libraries.

We are now ready to create a project-specific *Python* environment as well as to install all required packages. Note that, it's always recommended to create a separate *Python* environment for every new project so to avoid modifying system's default one. We can do so simply by running gist A.6 from our terminal.

```
#!/bin/bash

# remove old environment.
$ rm -rf env

# initialize new environment.
$ python3.7 -m venv env

# activate the environment.
$ source env/bin/activate

# install required installers.
$ pip install --upgrade pip setuptools

# install packages for Viz, CV, ML, DL, etc.
$ pip install \
    matplotlib==3.1.3 seaborn==0.10.0 pydot==1.4.1 \
    numpy==1.18.1 scipy==1.4.1 pandas==1.0.1 \
    scikit-learn==0.22.1 \
    tensorflow-cpu==2.1.1 keras==2.2.5 tensorflow-addons==0.10.0 \
    pillow==7.0.0 opencv-python==4.2.0.32 scikit-image==0.16.2 \
    jupyter==1.0.0 jupyterthemes==0.20.0
```

GIST A.6: Create Python environment and install packages.

The final step is, of course, to install *PyCharm*. We can do so by running gist A.7 from our terminal.

```
#!/bin/bash
$ wget https://jetbrains.com/python/pycharm-community-2020.1.1.tar.gz
$ tar -xzf pycharm-community-2020.1.1.tar.gz
$ rm pycharm-community-2020.1.1.tar.gz
$ cd pycharm-community-2020.1.1/bin/
$ bash pycharm.sh
```

GIST A.7: Install and start the PyCharm IDE.

If everything was done right up to here, we should be able to see something similar to figure A.1; the *PyCharm*, fully loaded with the project's source code.

Starting *Jupyter* is a slightly different story. Firstly, we don't have to install anything additional. If we look closely, we have already done this as part of a previous step. Secondly, the way to start *Jupyter* is simply by running gist A.8 from our terminal.

```
#!/bin/bash
```

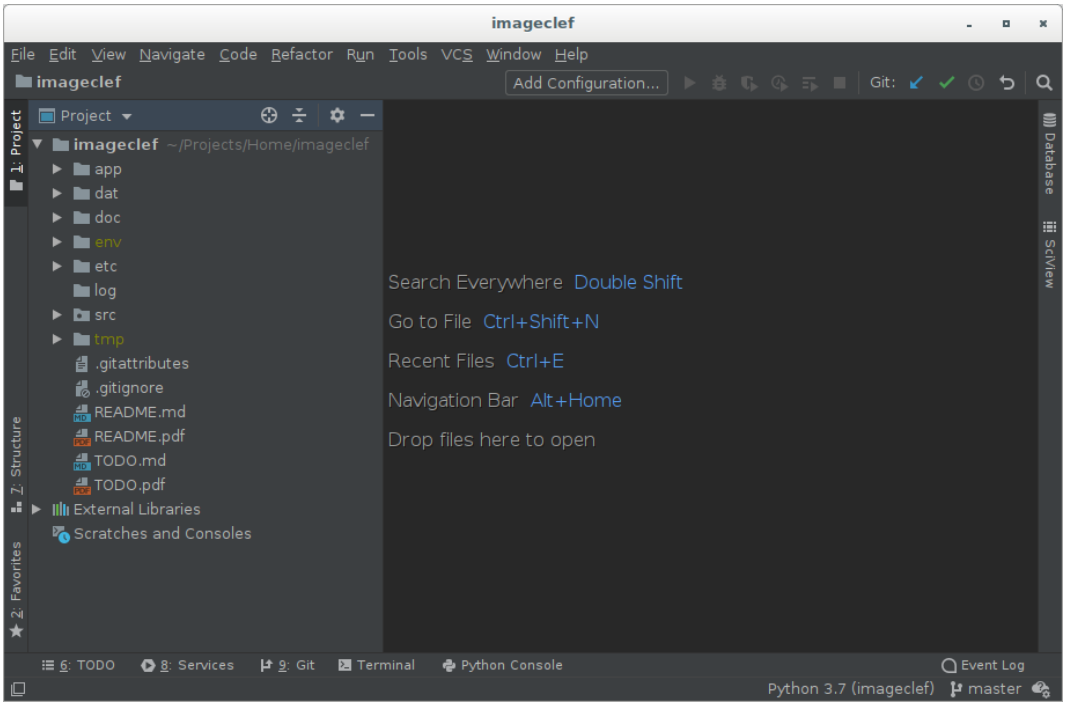

FIGURE A.1: The PyCharm IDE.

```
$ source env/bin/activate
$ rm -rf /home/$USER/.local/share/jupyter/
$ jupyter notebook
```

GIST A.8: Install and start the Jupyter Server.

Once the service is up and running, we should be able to see something similar to figure A.2 in our terminal. We simply click on the second *URL* and *Jupyter* will open up in our browser automatically.

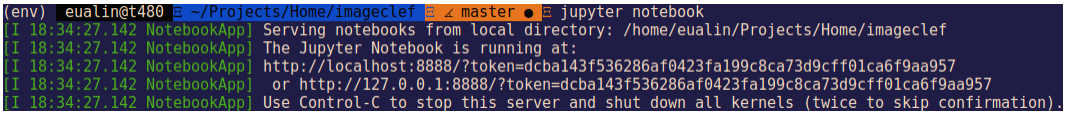

FIGURE A.2: The local Jupyter Server's address.

If everything was done right up to here, we should be able to see something similar to figure A.3; the *Jupyter*, fully loaded with the project's source code.

When we are done with our work on *Jupyter*, we can simply go on and close the browser tab, get back to the terminal and hit **CTRL + C** to stop the service, and finally type in **deactivate** to exit the *Python* environment.

We're all set! We can now start building *DL* applications in the **local** workstation!

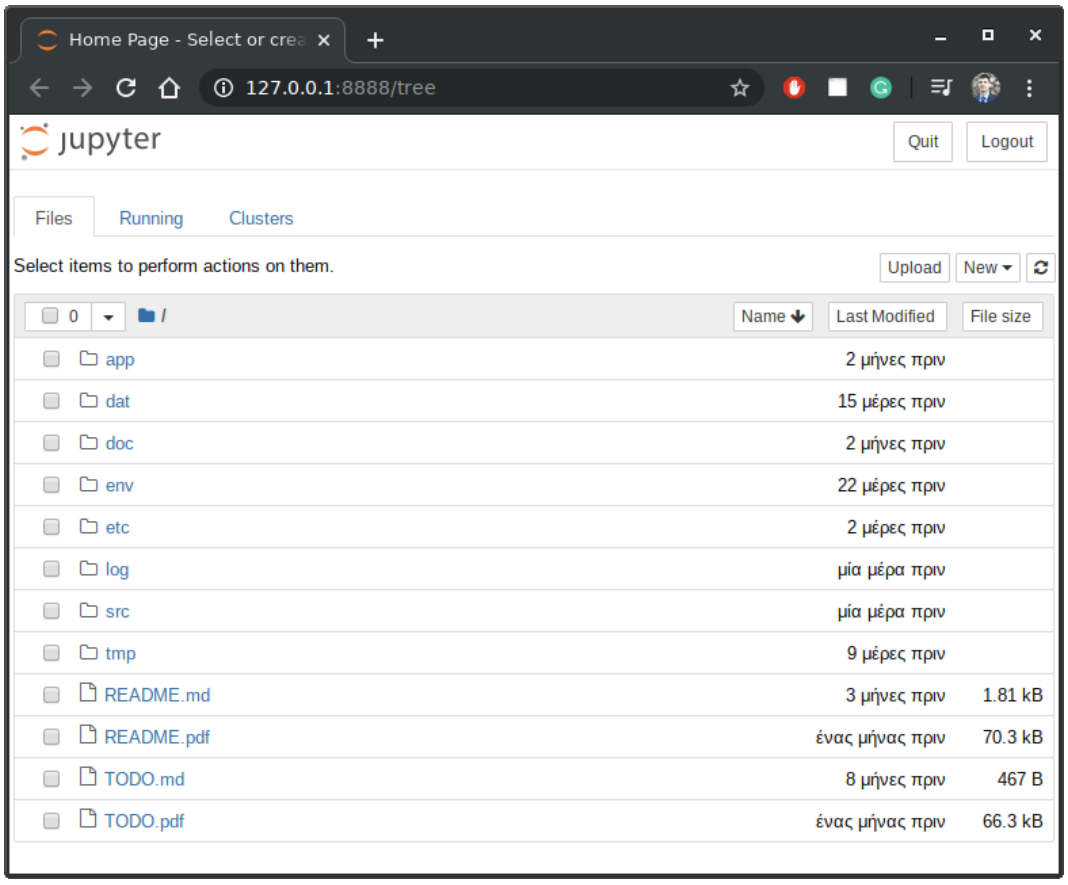

FIGURE A.3: The local Jupyter Notebook.

A.2 Setup Remote Instance

In this section, we will setup a *GPU* instance on *AWS EC2*. As pointed out earlier, many *DL* applications are so much computationally intensive that can take hours, days, or even weeks to finish with *CPUs*. However, running them on top of *GPUs* can speed up the process orders of magnitude.

Here, we will use a *p2.xlarge* instance from *US West (Oregon)* region, with a *CPU* of 4 cores, a *RAM* of 61 GiB, and a *GPU* of 11 GiB. Under the hood, the *p2.xlarge* instance utilizes one (out of two available) *Tesla K80 GPU* from *Nvidia*. The hourly cost for this instance is about $0.90, without considering costs for storage, inbound and outbound traffic, etc.

In our case, we were fortunate enough to receive a grant of $5,000 from *AWS Cloud Credits for Research* program. Obviously, without the *AWS* support (figure A.4), these resource-intensive experiments would never be possible.

For completeness sake, here is a copy of the acceptance letter:

"***Congratulations! Your application for the AWS Cloud Credits for Research program was successful! You have been awarded 5K in AWS credits for your project.***"

Now, let's move on with the instance setup. The whole process will take roughly about 10 minutes to complete or less.

FIGURE A.4: The AWS Cloud Credits for Research logo.

First of all, if not done already, we must create an account. We need to choose a support plan (the basic one will suffice) and provide the credit card information (no charges will be applied yet).

Next, we have to launch the *EC2* instance. Go to the *EC2 Management Console* and click on **Launch Instance** (figure A.5).

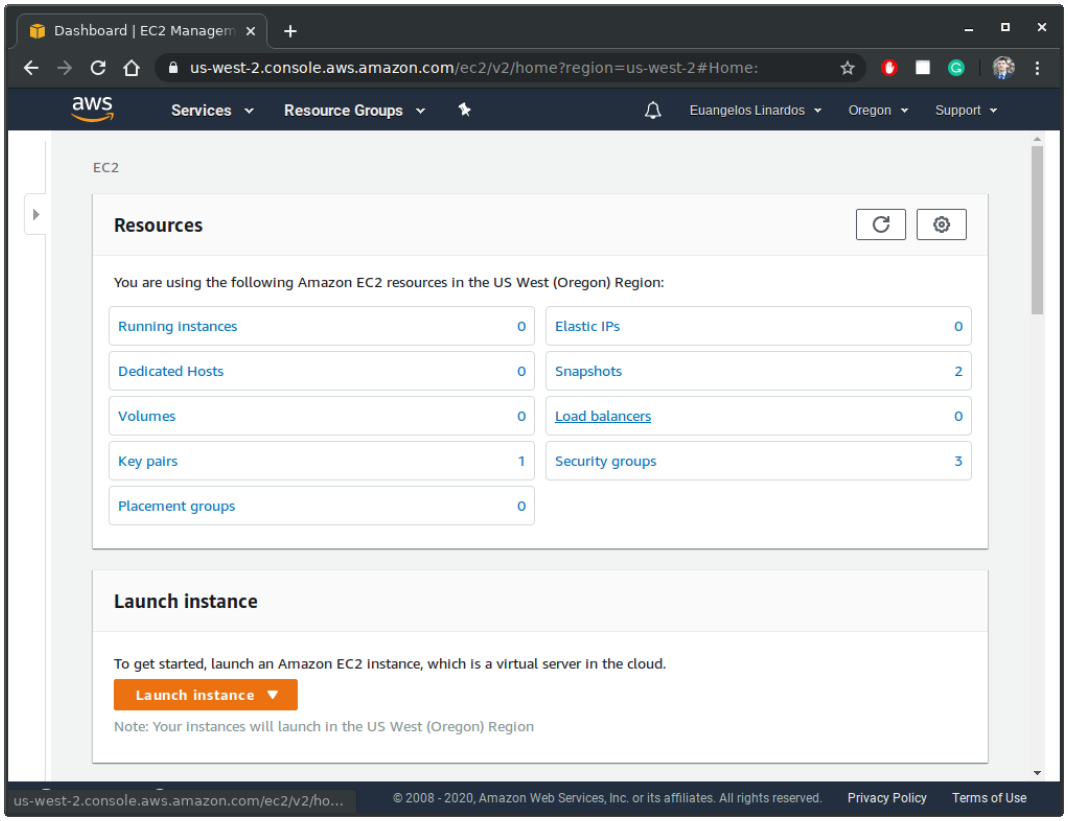

FIGURE A.5: The AWS Management Console interface.

Next, click on **AWS Marketplace** from the left-hand side, search for **Deep Learning AMI (Ubuntu 18.04) Version 28.1**, and hit **Select**. The selected *AMI* contains all the drivers, binaries, and files needed to train on a *GPU* instance (figure A.6).

In the next page we have to choose an instance type. First, we filter by **GPU Instances** only. Next, we select the **p2.xlarge** type and we click **Next** (figure A.7).

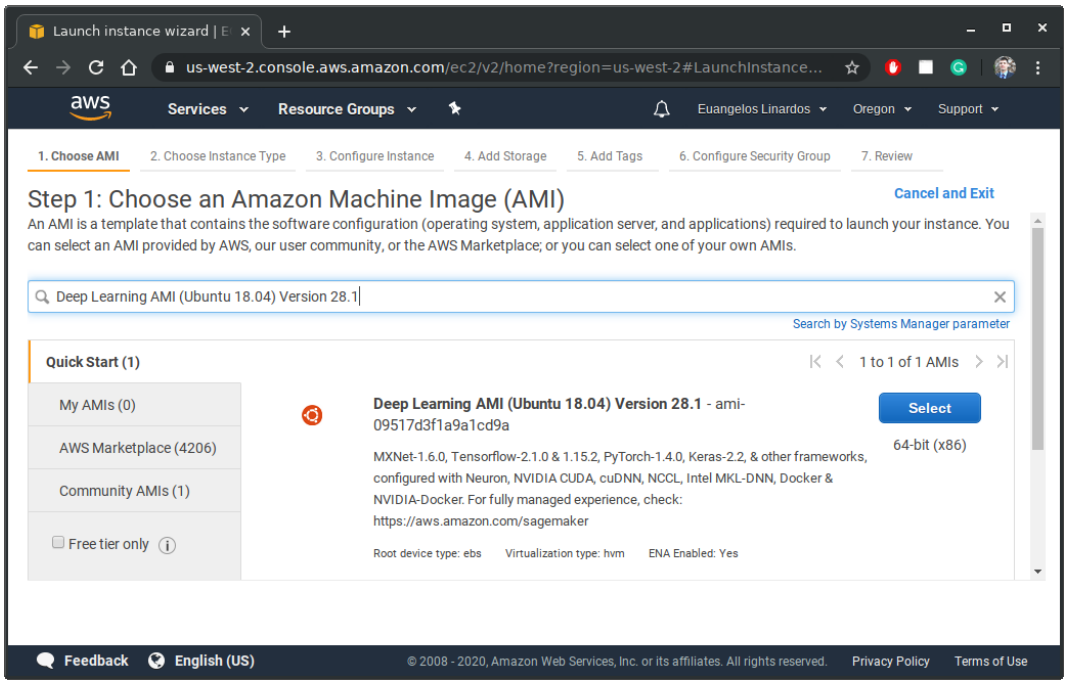

FIGURE A.6: Choose an AMI.

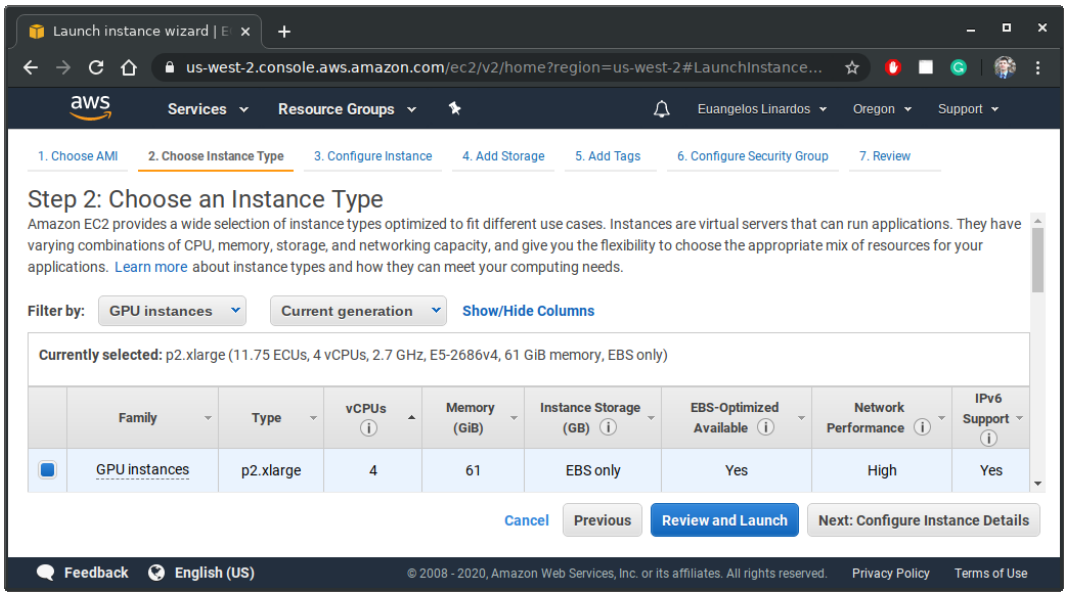

FIGURE A.7: Choose an instance type.

Skip the next three tabs, namely, **Configure Instance**, **Add Storage**, and **Add Tags**, and head toward the **Configure Security Group** tab to setup a few security rules accordingly (figure A.8).

Note that if we allow **Anywhere** for a port, then, literally, anyone will be able to listen to that port of our instance. If at all possible, we should consider restricting access to a specific *IP* address only. But if our *IP* changes constantly, then, this is not a viable choice. If we are going to leave access open to any *IP*, then, at least, we should be cautious to not store sensitive data to our instance.

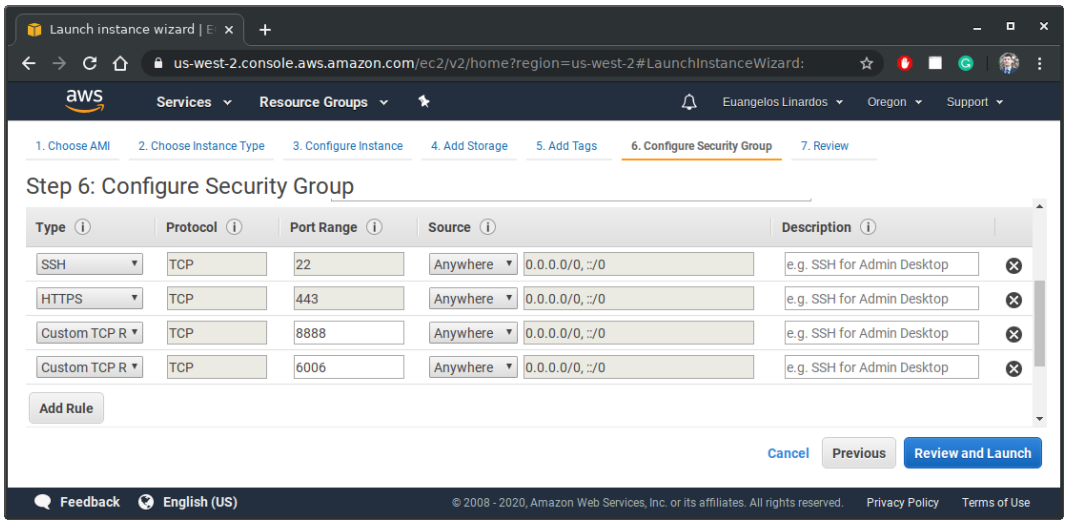

FIGURE A.8: Configure the security group.

Finally, we should click on **Review and Launch** to actually launch the instance. We'll need to specify an authentication *key pair* to be able to access the instance via *SSH* later on (figure A.9).

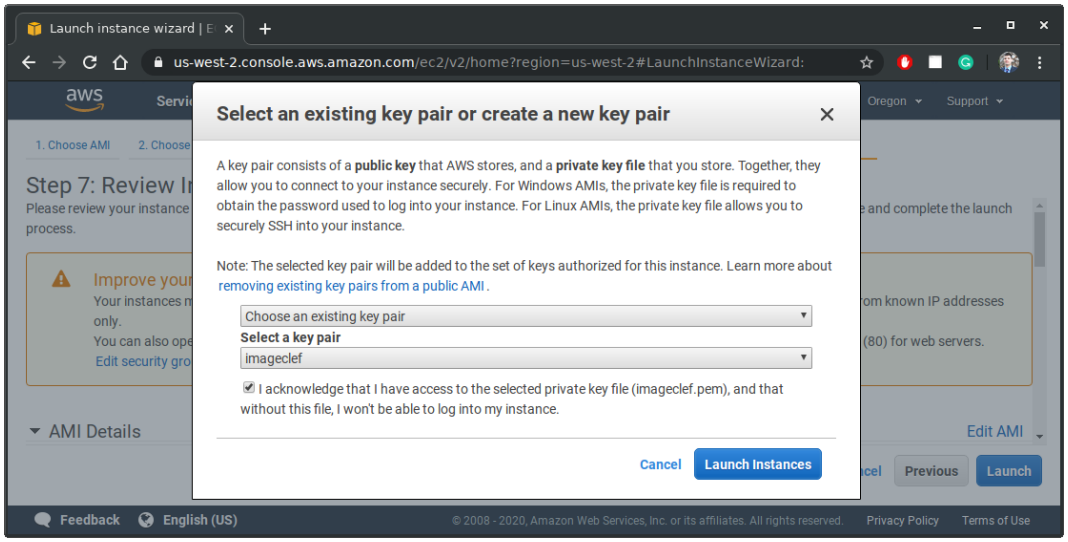

FIGURE A.9: Select a key pair.

We click **Review and Launch** once more, and we're done! Note that, from this point on, *AWS* will charge us for running the instance. So, we should always remember to stop the instance when we are not using it, otherwise, the instance will keep running and we'll wind up with a hefty bill! *AWS* charges primarily for running instances, so most of the charges will cease once we stop the instance. However, there are smaller storage charges that continue to accumulate until we terminate the instance.

Allow a minute (or two) for the instance to launch. We will know that it's ready for use when the **Status Checks** indicates that all checks have passed. Now let's switch to the **Description** tab on the *EC2* dashboard and make a note of the **IPv4 Public IP**

address. We will need this info during next step in order to access the instance via *SSH* (figure A.10).

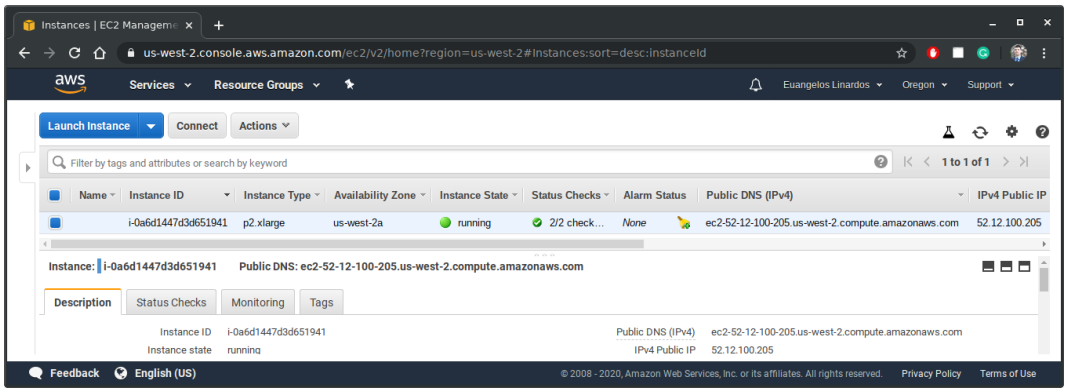

FIGURE A.10: List of currently running instances.

Next, go to the terminal and connect to the *EC2* instance via *SSH*. But first, navigate to the location where we have stored the *key pair* from a previous step and type in the commands of gist A.9.

```
#!/bin/bash
$ chmod 400 imageclefmed.pem && ssh -i imageclefmed.pem ubuntu@52
    .12.100.205
```

GIST A.9: SSH instance.

The final step is to configure and run the *Jupyter Notebook Server* on *EC2*. After we have accessed the **remote** instance via *SSH*, we need to follow the steps of figure A.11.

```
ubuntu@ip-172-31-55-154:~$ jupyter notebook password
Enter password:
Verify password:
[NotebookPasswordApp] Wrote hashed password to /home/ubuntu/.jupyter/jupyter_notebook_config.json
ubuntu@ip-172-31-55-154:~$ mkdir ssl
ubuntu@ip-172-31-55-154:~$ cd ssl
ubuntu@ip-172-31-55-154:~/ssl$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.key -out mycert.pem
Generating a RSA private key
.....................................................+++++
...........+++++
writing new private key to 'mykey.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GR
State or Province Name (full name) [Some-State]:Attika
Locality Name (eg, city) []:Athens
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ImageCLEFmed
Organizational Unit Name (eg, section) []:AI Lab
Common Name (e.g. server FQDN or YOUR name) []:Euangelos Linardos
Email Address []:icsdm117022@icsd.aegean.gr
ubuntu@ip-172-31-55-154:~/ssl$ jupyter notebook --certfile=~/ssl/mycert.pem --keyfile ~/ssl/mykey.key
[I 07:07:19.469 NotebookApp] Using EnvironmentKernelSpecManager...
[I 07:07:19.470 NotebookApp] Started periodic updates of the kernel list (every 3 minutes).
[I 07:07:19.652 NotebookApp] Writing notebook server cookie secret to /home/ubuntu/.local/share/jupyter/runtime/notebook_cookie_secret
[I 07:07:25.197 NotebookApp] Loading IPython parallel extension
[I 07:07:29.234 NotebookApp] JupyterLab extension loaded from /home/ubuntu/anaconda3/lib/python3.7/site-packages/jupyterlab
[I 07:07:29.234 NotebookApp] JupyterLab application directory is /home/ubuntu/anaconda3/share/jupyter/lab
[I 07:07:32.737 NotebookApp] [nb_conda] enabled
[I 07:07:32.738 NotebookApp] Serving notebooks from local directory: /home/ubuntu/ssl
[I 07:07:32.738 NotebookApp] The Jupyter Notebook is running at:
[I 07:07:32.738 NotebookApp] https://localhost:8888/
[I 07:07:32.738 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

FIGURE A.11: Configure and start the remote Jupyter Server.

Then, we should open a second shell in the host machine and type in the commands of gist A.10, the purpose of which is to forward the **local** port 8888 to port 8888 in the **remote** instance.

```
#!/bin/bash
$ ssh -i imageclefmed.pem -N -f -L 8888:localhost:8888 ubuntu@52
    .12.100.205
```

GIST A.10: Port forwarding.

Finally, we open a browser and we navigate to the **local** address we are forwarding to the **remote** notebook process (i.e., *http://127.0.0.1:8888/*). There we will see a security warning due to the fact that the *SSL* certificate we generated earlier is not verified by a trusted organization. Except that, we created that certificate on our behalf and so it is for sure trusted. Therefore, we can simply click on **Advanced** and then again on **Proceed** as illustrated in figures A.12 and A.13.

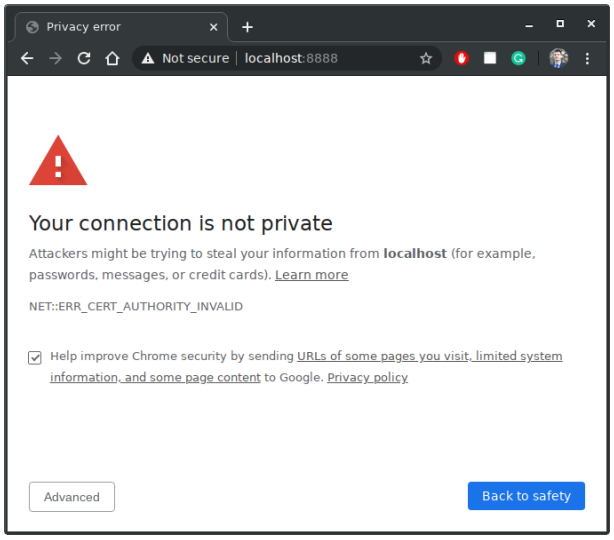

FIGURE A.12: Invalid certificate.

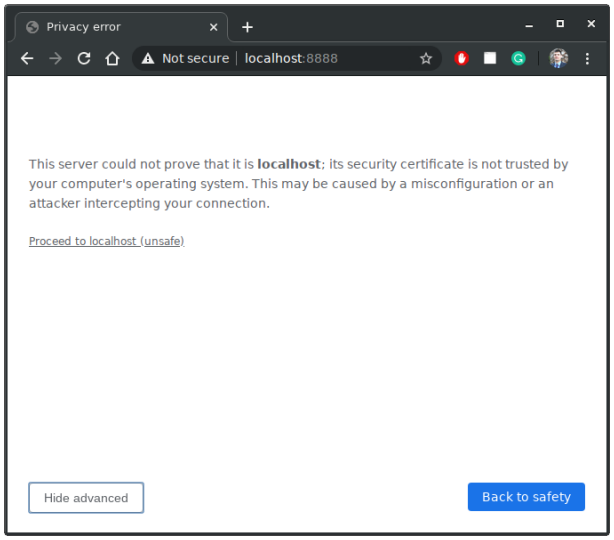

FIGURE A.13: Accept certificate.

At this point, we should have been prompted to enter the *Jupyter* password. So, let's type it in and click **Log In** to access the notebook dashboard as illustrated in figure A.14.

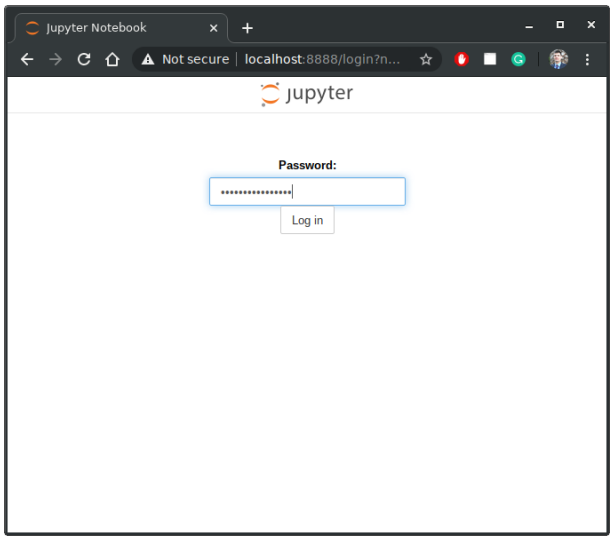

FIGURE A.14: Enter password.

If everything was done right up to here, we should be able to see something similar to figure A.15; the *Jupyter*.

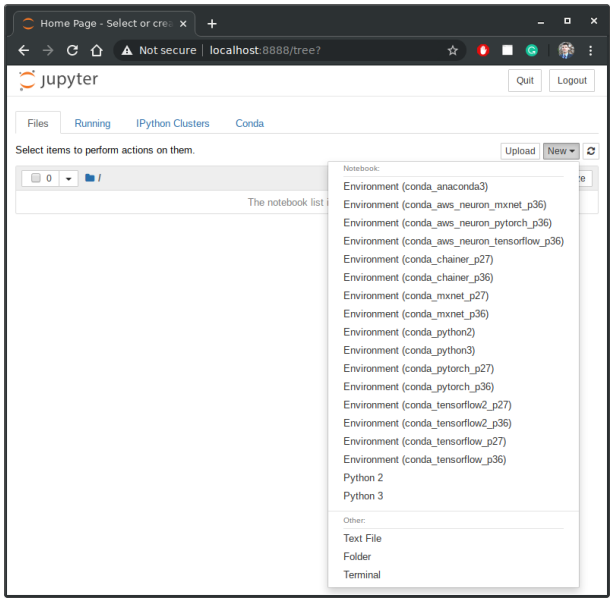

FIGURE A.15: The remote Jupyter Notebook.

We're all set! We can now start building *DL* applications in the **remote** instance!

Bibliography

Bates, David W. et al. (2001). "White Paper: Reducing the Frequency of Errors in Medicine Using Information Technology". In: *Journal of the American Medical Informatics Association* 8.4, pp. 299–308. DOI: 10.1136/jamia.2001.0080299. URL: https://doi.org/10.1136/jamia.2001.0080299.

Chatfield, Ken et al. (2014). "Return of the Devil in the Details: Delving Deep Into Convolutional Nets". In: ed. by Michel François Valstar, Andrew P. French, and Tony P. Pridmore. URL: http://www.bmva.org/bmvc/2014/papers/paper054/index.html.

Chen, Shang-Fu et al. (2018a). "Order-free RNN With Visual Attention for Multi-label Classification". In: ed. by Sheila A. McIlraith and Kilian Q. Weinberger, pp. 6714–6721. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16114.

Chen, Tianshui et al. (2018b). "Recurrent Attentional Reinforcement Learning for Multi-label Image Recognition". In: ed. by Sheila A. McIlraith and Kilian Q. Weinberger, pp. 6730–6737. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16654.

Chen, Zhao-Min et al. (2019). "Multi-label Image Recognition With Graph Convolutional Networks". In: pp. 5177–5186. DOI: 10.1109/CVPR.2019.00532. URL: http://openaccess.thecvf.com/content_CVPR_2019/html/Chen_Multi-Label_Image_Recognition_With_Graph_Convolutional_Networks_CVPR_2019_paper.html.

Deng, Jia et al. (2009). "ImageNet: A Large-scale Hierarchical Image Database". In: pp. 248–255. DOI: 10.1109/CVPR.2009.5206848. URL: https://doi.org/10.1109/CVPR.2009.5206848.

Durand, Thibaut, Nazanin Mehrasa, and Greg Mori (2019). "Learning a Deep ConvNet for Multi-label Classification With Partial Labels". In: pp. 647–657. DOI: 10.1109/CVPR.2019.00074. URL: http://openaccess.thecvf.com/content_CVPR_2019/html/Durand_Learning_a_Deep_ConvNet_for_Multi-Label_Classification_With_Partial_Labels_CVPR_2019_paper.html.

Eickhoff, Carsten et al. (2017). "Overview of ImageCLEFcaption 2017 - Image Caption Prediction and Concept Detection for Biomedical Images". In: CEUR Workshop Proceedings 1866. Ed. by Linda Cappellato et al. URL: http://ceur-ws.org/Vol-1866/invited_paper_7.pdf.

Esteva, Andre et al. (2017). "Dermatologist-level Classification of Skin Cancer With Deep Neural Networks". In: *Nature* 542.7639, pp. 115–118. DOI: 10.1038/nature21056. URL: https://doi.org/10.1038/nature21056.

Everingham, Mark et al. (2010). "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2, pp. 303–338. DOI: 10.1007/s11263-009-0275-4. URL: https://doi.org/10.1007/s11263-009-0275-4.

Ge, Weifeng, Sibei Yang, and Yizhou Yu (2018). "Multi-evidence Filtering and Fusion for Multi-label Classification, Object Detection and Semantic Segmentation Based on Weakly Supervised Learning". In: pp. 1277–1286. DOI: 10.1109/CVPR.2018.

00139. URL: http://openaccess.thecvf.com/content_cvpr_2018/html/Ge_Multi-Evidence_Filtering_and_CVPR_2018_paper.html.

George, Marian and Christian Floerkemeier (2014). "Recognizing Products: A Per-exemplar Multi-label Image Classification Approach". In: Lecture Notes in Computer Science 8690. Ed. by David J. Fleet et al., pp. 440–455. DOI: 10.1007/978-3-319-10605-2_29. URL: https://doi.org/10.1007/978-3-319-10605-2_29.

Gong, Yunchao et al. (2014). "Deep Convolutional Ranking for Multilabel Image Annotation". In: ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1312.4894.

Herrera, Alba Garcia Seco de et al. (2018). "Overview of the ImageCLEF 2018 Caption Prediction Tasks". In: CEUR Workshop Proceedings 2125. Ed. by Linda Cappellato et al. URL: http://ceur-ws.org/Vol-2125/invited_paper_4.pdf.

Huang, Gao et al. (2017). "Densely Connected Convolutional Networks". In: pp. 2261–2269. DOI: 10.1109/CVPR.2017.243. URL: https://doi.org/10.1109/CVPR.2017.243.

Katsios, Dimitris and Ergina Kavallieratou (2017). "Concept Detection on Medical Images Using Deep Residual Learning Network". In: CEUR Workshop Proceedings 1866. Ed. by Linda Cappellato et al. URL: http://ceur-ws.org/Vol-1866/paper_122.pdf.

Kougia, Vasiliki, John Pavlopoulos, and Ion Androutsopoulos (2019). "AUEB NLP Group at ImageCLEFmed Caption 2019". In: CEUR Workshop Proceedings 2380. Ed. by Linda Cappellato et al. URL: http://ceur-ws.org/Vol-2380/paper_136.pdf.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification With Deep Convolutional Neural Networks". In: ed. by Peter L. Bartlett et al., pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

Lee, June-Goo et al. (2017). "Deep Learning in Medical Imaging: General Overview". In: *Korean Journal of Radiology* 18.4, pp. 570–584. URL: https://pubmed.ncbi.nlm.nih.gov/28670152/.

Litjens, Geert et al. (2017). "A Survey on Deep Learning in Medical Image Analysis". In: *Medical Image Analysis* 42, pp. 60–88. DOI: 10.1016/j.media.2017.07.005. URL: https://doi.org/10.1016/j.media.2017.07.005.

Oakden-Rayner, Luke (2019). "Exploring Large Scale Public Medical Image Datasets". In: *CoRR* abs/1907.12720. arXiv: 1907.12720. URL: http://arxiv.org/abs/1907.12720.

Oquab, Maxime et al. (2014). "Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks". In: pp. 1717–1724. DOI: 10.1109/CVPR.2014.222. URL: https://doi.org/10.1109/CVPR.2014.222.

Pelka, Obioma, Felix Nensa, and Christoph M. Friedrich (2018). "Adopting Semantic Information of Grayscale Radiographs for Image Classification and Retrieval". In: 2. Ed. by Sheldon Wiebe et al., pp. 179–187. DOI: 10.5220/0006732301790187. URL: https://doi.org/10.5220/0006732301790187.

Pelka, Obioma et al. (2018). "Radiology Objects in COntext (ROCO): A Multimodal Image Dataset". In: Lecture Notes in Computer Science 11043. Ed. by Danail Stoyanov et al., pp. 180–189. DOI: 10.1007/978-3-030-01364-6_20. URL: https://doi.org/10.1007/978-3-030-01364-6_20.

Pelka, Obioma et al. (2019). "Overview of the ImageCLEFmed 2019 Concept Detection Task". In: CEUR Workshop Proceedings 2380. Ed. by Linda Cappellato et al. URL: http://ceur-ws.org/Vol-2380/paper_245.pdf.

Pinho, Eduardo and Carlos Costa (2018). "Feature Learning With Adversarial Networks for Concept Detection in Medical Images: UA.PT Bioinformatics at ImageCLEF 2018". In: CEUR Workshop Proceedings 2125. Ed. by Linda Cappellato et al. URL: http://ceur-ws.org/Vol-2125/paper_139.pdf.

Rajpurkar, Pranav et al. (2017). "CheXNet: Radiologist-level Pneumonia Detection on Chest X-Rays With Deep Learning". In: *CoRR* abs/1711.05225. arXiv: 1711.05225. URL: http://arxiv.org/abs/1711.05225.

Rajpurkar, Pranav et al. (2018). "Deep Learning for Chest Radiograph Diagnosis: A Retrospective Comparison of the CheXNeXt Algorithm to Practicing Radiologists". In: *PLOS Medicine* 15.11, pp. 1–17. DOI: 10.1371/journal.pmed.1002686. URL: https://doi.org/10.1371/journal.pmed.1002686.

Ren, Shaoqing et al. (2015). "Faster R-CNN: Towards Real-time Object Detection With Region Proposal Networks". In: ed. by Corinna Cortes et al., pp. 91–99. URL: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.

Roberts, Richard J. (2001). "PubMed Central: The GenBank of the Published Literature". In: *Proceedings of the National Academy of Sciences of the United States of America* 98.2, pp. 381–382. DOI: 10.1073/pnas.98.2.381. URL: https://doi.org/10.1073/pnas.98.2.381.

Russakovsky, Olga et al. (2015). "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3, pp. 211–252. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y.

Schmidhuber, Jürgen (2015). "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61, pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003. URL: https://doi.org/10.1016/j.neunet.2014.09.003.

Shin, Hoo-Chang et al. (2016). "Learning to Read Chest X-Rays: Recurrent Neural Cascade Model for Automated Image Annotation". In: pp. 2497–2506. DOI: 10.1109/CVPR.2016.274. URL: https://doi.org/10.1109/CVPR.2016.274.

Wang, Xiaosong et al. (2017). "ChestX-Ray8: Hospital-scale Chest X-Ray Database and Benchmarks on Weakly-supervised Classification and Localization of Common Thorax Diseases". In: pp. 3462–3471. DOI: 10.1109/CVPR.2017.369. URL: https://doi.org/10.1109/CVPR.2017.369.

Wang, Xuwen et al. (2018). "ImageSem at ImageCLEF 2018 Caption Task: Image Retrieval and Transfer Learning". In: CEUR Workshop Proceedings 2125. Ed. by Linda Cappellato et al. URL: http://ceur-ws.org/Vol-2125/paper_129.pdf.

Wei, Yunchao et al. (2016). "HCP: A Flexible CNN Framework for Multi-label Image Classification". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.9, pp. 1901–1907. DOI: 10.1109/TPAMI.2015.2491929. URL: https://doi.org/10.1109/TPAMI.2015.2491929.