First contact with Deep Learning

Practical introduction with Keras

Jordi Torres

WATCH THIS SPACE

First contact with Deep Learning Practical introduction with Keras

Jordi Torres

WATCH THIS SPACE

WATCH THIS SPACE COLLECTION - BARCELONA
2018

First contact with Deep Learning, Practical introduction with Keras

Jordi Torres
WATCH THIS SPACE collection – Barcelona: Book 5
Kindle Direct Publishing.
www.amazon.com

Cover: Background (Roser Bellido). Picture (BSC-CNS using DL style transfer form algorithmia.com)

ISBN 978-1-983-21155-3

Fisrt edition: July 2018.

©Jordi Torres
http://www.JordiTorres.Barcelona
Universitat Politècnica de Catalunya - UPC Barcelona Tech
Campus Nord, mòdul C6
Jordi Girona 1-3
08034 Barcelona

Writer's Coach: Ferran Julià Massó

granted, provided the authorship and licensing information here is displayed

To all my students at UPC (*)

(*)

According to Professor Howard Gardner, the older you are, the harder it is to adapt your life to a new idea and the easier it is to adapt that new idea to your comfortable life without changing it. That's why I go to class to unlearn from myself and learn from my young students that I have in class.

Contents:

Foreword

In 1953, Isaac Asimov published *Second Foundation*, the third book of the *Foundation* series (or the thirteenth according to other sources, this is a subject of debate). In *Second Foundation* Arkady Darell, one of the main characters of the final part of the saga, appears for the first time. In his first scene, Arkady, who is 14 years old, is doing his homework. Specifically, an essay entitled "Seldon Plan". To do the writing, Arkady is using a "transcription machine", a device that converts his voice into written words. Arkady speaks into the machine and it prints out his words in a fancy calligraphic font. This type of device, which for Isaac Asimov was science fiction in 1953, is available in most of our smartphones, and Deep Learning is one of the factors responsible for it, in addition to advances in technology.

On the other hand, the computing technology currently available is another of the key factors responsible for us having these applications. Nowadays we have GPUs (Graphics Processor Units), which only cost around 100 euros, which would be on the Top500 list a few years ago (competing with machines that cost millions of euros). The GPUs were designed to facilitate the programming of video games, but a combination of small changes (unified shaders, shared memory, memory access instructions, Tensor Cores ... it may not be so small!), and the emergence of new programming models (CUDA, OpenCL, OpenACC), have facilitated the efficient use of GPUs in general-purpose applications, including Deep Learning. In addition, considering that every year more than 1,000 million smartphones are sold (all of them have a GPU) and that the videogame business is very attractive, it is guaranteed that the technological improvement of GPUs will continue for a long time.

I have known Jordi for more than 30 years; He has always been very concerned about the latest technological advances, not only in our area of knowledge, *Computer Architecture*, but also in broader issues that we could include in what is known as *Computer Science*. Jordi is thinking about research, but also about transferring that knowledge to our students (this, and no other, is the ultimate goal of research at the university). Considering this, it is not surprising that he has embarked on publishing a series of books dedicated to Deep Learning. These tools are changing the way computer problems are addressed and are opening the spectrum of what a computer can do.

And what will the future hold? To be better informed, we can start reading this book, but in the next few years, we will have a series of revolutionary applications closely related to Deep Learning: autonomous cars, natural language processing, machine translation...

Those who will have many more difficulties will be the writers of science fiction: It will be very difficult for them to imagine new devices that have not already been designed by current engineers.

Agustín Fernández
Vice-rector for Digital Transformation
Universitat Politècnica de Catalunya • BarcelonaTech

Preface: the passion to teach

“Education is the most powerful weapon which you can use to change the world.”

Nelson Mandela

Those who know me, know that something that has always motivated me, is to contribute by being the proverbial spark that lights the fire and awakens minds, in order to be prepared for the changes that our society faces, in the wake of the impact of technologies such as artificial intelligence in general and Deep Learning in particular.

I've always been interested in next-generation technology and its impact, and that's why artificial intelligence and its relationship with technologies such as Cloud Computing, Big Data or supercomputing (high performance computing) have caught my interest. These are areas in which I have been researching and teaching for 30 years.

Undoubtedly, technological advances in Artificial Intelligence, along with the other aforementioned technologies, are already here and allow us to build a

society that improves people's lives. On the other hand, it is also true that in the near future these technologies will present other uncertainties.

I am however convinced , that together, as a society, we can find solutions to new problems we come across, that are brought about by these new technologies. For this reason, it is key that each and every one of us ~~who~~ working in the world of technology has a better understanding of these new topics that are revolutionizing IT, in order that we can use them correctly and also be able to explain them to society.

This book is an English translation of the recently published book "*Deep Learning - Introducción práctica con Keras*"[1] that I wrote during the free time that my academic and research activity allowed me.

The book was initially designed to support my classes at the *Facultat d'Informàtica de Barcelona*, the faculty of Computer Science of the *Universitat Politècnica de Catalunya – Barcelona Tech* (UPC).

And as I have done with all the books in the collection WATCH THIS SPACE, I have openly published all its content on my personal website https://www.JordiTorres.org/DeepLearning

I'm excited to share it with all my students and with the entire IT community interested in having a first contact with this technology and help them to start on their own in the programming of Deep Learning. I hope this introductory book will help the reader interested in starting their adventure in this very interesting field. If the reader finds the content useful and prefers to opt for a paper copy or eBook copy, it can be purchased on *Amazon*.

As the title suggests, this book is only intended to be a practical introduction to Deep Learning, currently one of the most active areas in the field of Artificial Intelligence. It is not meant as an exhaustive treatise on the subject. This book is aimed at the reader who has knowledge in programming but has not yet had the opportunity to start these new key concepts in computing, and

wants to discover some of the possibilities and more cutting-edge aspects of this technology.

I advise you that the book will be an invitation to use your computer keyboard while you are learning: we call it "to learn by doing", and from my experience as a professor at the UPC, it is an approach that works very well with engineers that want to take a quick look at a new technology.

For this reason, the book will have an eminently practical orientation. Therefore the theoretical-mathematical part will be reduced as much as possible, although it is strictly necessary to present some theoretical details to offer a solid knowledge to the reader. This means that the pages will be interspersed with theoretical and practical knowledge, complementary to the learning process.

Before finishing this preface, let me say thanks for reading this book! This fact alone makes me happy and justifies my effort in writing it. Those who know me well, know that teaching (and education in general) is one of my passions and keeps me energized and motivated.

As my dear mother would sometimes reminds me, when I was young I used to say that when I grew up I wanted to teach, to be a teacher. Well, here I am, my dream came true!

Jordi Torres i Viñals. July 15, 2018.

If the reader needs to contact the author in relation to the content of this book or wants to send him their comments, they can do so via email: Jordi.Torres@DeepLearning.Barcelona

Preliminary considerations

In this book, the reader will find a guide to understanding the basics of Deep Learning with the help of the Keras library, which they will learn to use with the aim of developing and evaluating Deep Learning models. Although Deep Learning is based on fascinating mathematics, it is not strictly necessary in order to start, or even to create projects that generate value for the company, thanks to Python libraries such as Keras.

Therefore, this book will focus on practical issues to show the reader the exciting world that can be opened up with the use of Deep Learning. It is important to keep in mind that we can only examine a small part because it is impossible to show its total scope in a single book. Just keeping abreast of the latest research, technologies or tools that are appearing is an almost impossible mission, like drinking from a fire hose

Audience for this book

This is an introductory work, initially designed to support my teaching duties at the UPC to students of computer engineering with little or no previous knowledge of Machine Learning. But in turn, this book can also be useful to engineers who have left the classrooms and are working but need, or simply want to, learn about this topic.

Therefore the book is written in the form of "short distance" with the reader, as if we were in one of my classes at the UPC. To facilitate this, I will introduce the knowledge and concepts sequentially, trying to involve the reader by requiring them to always have the keyboard in front of them and testing what I am explaining.

However, this book is not for everyone, and it is important that the reader has

the right expectations; In this book you will not find explanations about the fundamental theory of artificial neural networks, nor details of how the algorithms related to the subject work internally. Therefore, it is not recommended for those who are already programming neural networks in some other environment that is not Keras. Also, if this library is their field of interest, perhaps this book is way too long to start with the Keras library.

.

How this book is organized

The book is organized into chapters that must be read in order, as they guide the reader and gradually introduce them to the essential knowledge to follow the practical examples, whilst trying to be as concise as possible. Being an introductory book, I consider that this approach is much more useful than a more formal one. In spite of everything, I have tried to make sure that the index of the book expresses a reasonably ordered map of the main concepts of the area.

The book follows the same formula as the book I wrote about TensorFlow, *Hello World en TensorFlow*, in January 2016 which was well received. Therefore, I have decided to partially follow its approach. But before starting with the first chapter I have allowed myself to introduce how I started to investigate this topic and why I consider that the main trigger of this resurrection of Artificial Intelligence is due to Supercomputing.

The first chapter contains a motivation for the topic in order to contextualize the reader as to where we are currently in this field.

In chapter two, using a case study, the basic concepts of a neural network are explained and densely connected neural networks are introduced.

Next, in chapter three, we move on to a more practical level with the well

known MNIST digit recognition example, showing its implementation with Keras.

In chapter four, we present how the learning process of a neural network is carried out, going into some of its most important components.

In chapter five the reader is invited to practice some of the knowledge previously acquired with *TensorFlow playground*, a tool offered by Google.

In chapter six, with the reader already prepared with an important basis of how neural networks work, we present and implement *Convolutional Neural Networks*, one of the most popular families of neural networks at this time.

This book is accompanied by a repository code on the GitHub where the reader can find the examples presented.

Assumptions of this book

As mentioned earlier, this work is intended as an introduction; it is not necessary for the reader to be an expert in Python. Obviously, it is necessary to have programming knowledge and an interest in learning autonomously the Python details when the code shown in the examples is not understood.

Nor is it necessary to be an expert in Machine Learning, but it is clear that it can be very useful to know basic concepts in the field. Only the basic knowledge of mathematics is assumed (any student of any of the baccalaureate of the scientific-technical branch). Throughout the chapters, the most important concepts of Machine Learning that may be required will be presented very briefly.

Why Keras?

Keras[2] is the recommended library for beginners, since its learning curve is very smooth compared to others, and at the moment it is one of the popular

middleware to implement neural networks.

Keras is a Python library that provides, in a simple way, the creation of a wide range of Deep Learning models using as backend other libraries such as TensorFlow, Theano or CNTK. It was developed and maintained by François Chollet[3], an engineer from Google, and his code has been released under the permissive license of MIT.

Personally, I value the austerity and simplicity that this programming model presents, without adornments and maximizing readability; it allows us to express neural networks in a very modular way, considering a model as a sequence (or a *graph* if it is more advanced models that we will not deal with in this book). Last but not least, I think it's a great success to have opted to use the Python programming language. For all these reasons, I have decided to use Keras in this book.

Jupyter notebook

Keras is currently included in Tensorflow package, but can also be used as a Python library. To start in the subject I consider that this second option is the most appropriate and for this reason my proposal will be to use Jupyter[4] since it is a very widespread and very easy to use development environment. At the end of this chapter, we will present how to access and fine-tune our work environment to execute the examples of code presented in the book.

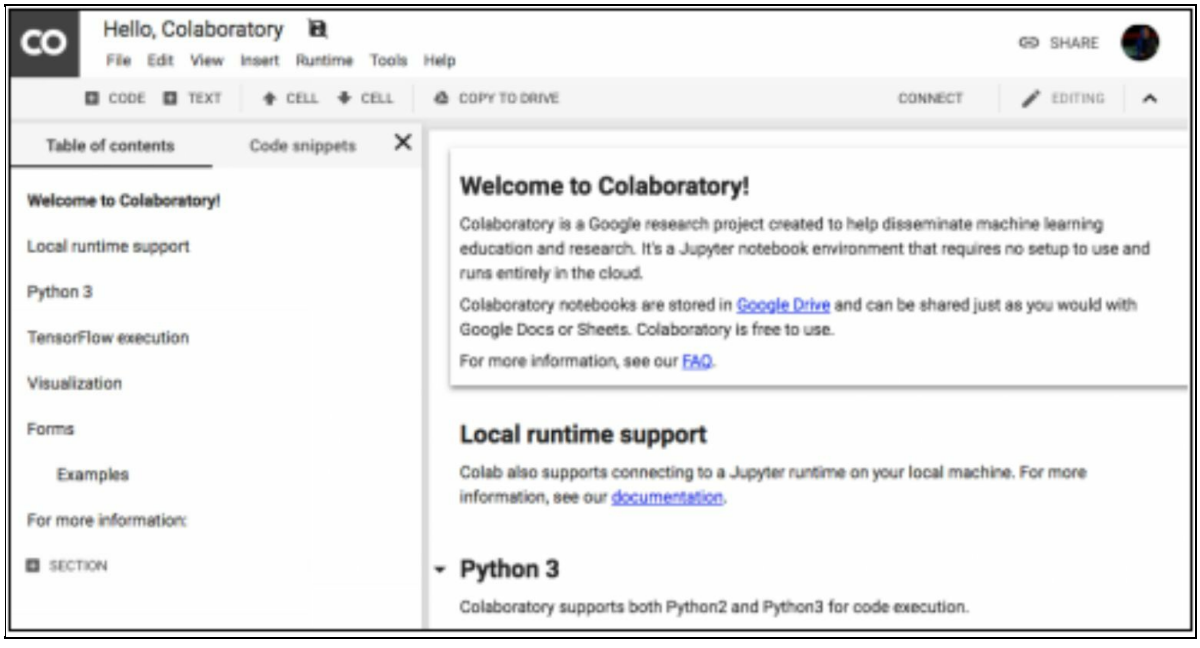
Jupyter provides great versatility to share parts of codes with annotations through the comfort and platform independence offered by a web browser. For this reason, notebooks are often used to develop neural networks in the community of scientists and data engineers. A notebook is a file generated by Jupyter Notebook or Jupyter Lab that can be edited from a web browser, allowing you to mix Python code execution with annotations.

The code in this book is available in the form of notebooks in the GitHub[5]

of the book, although this can be run as a normal program in Python if the reader so wishes.

Enrivonment set up

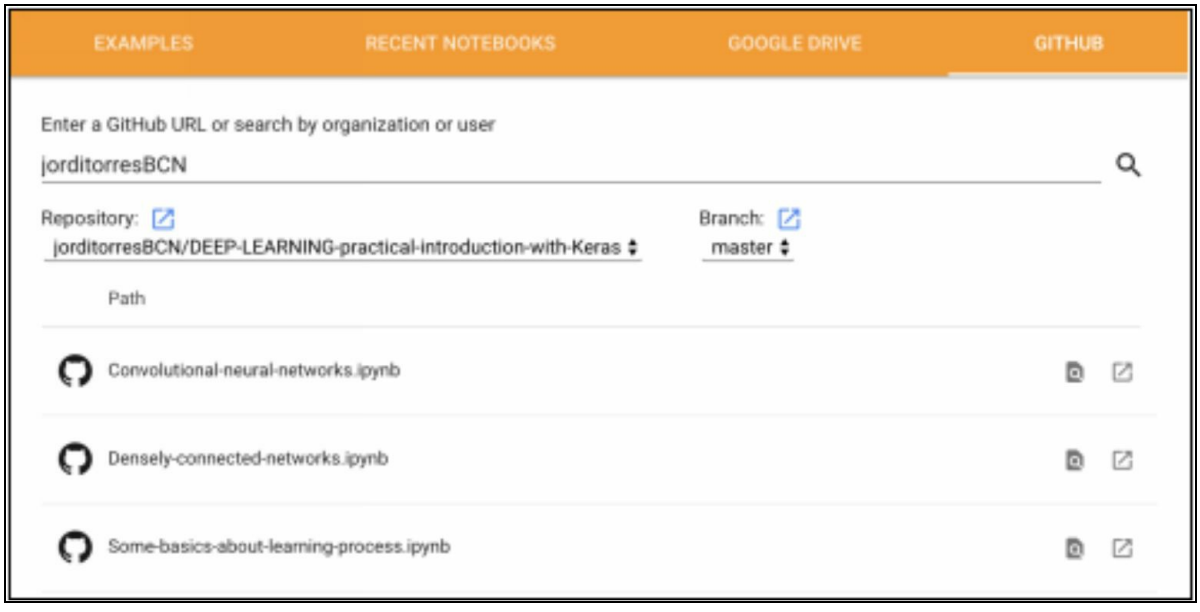
In this book, we will use the *Colaboratory*[6] environment (Colab) offered by Google.



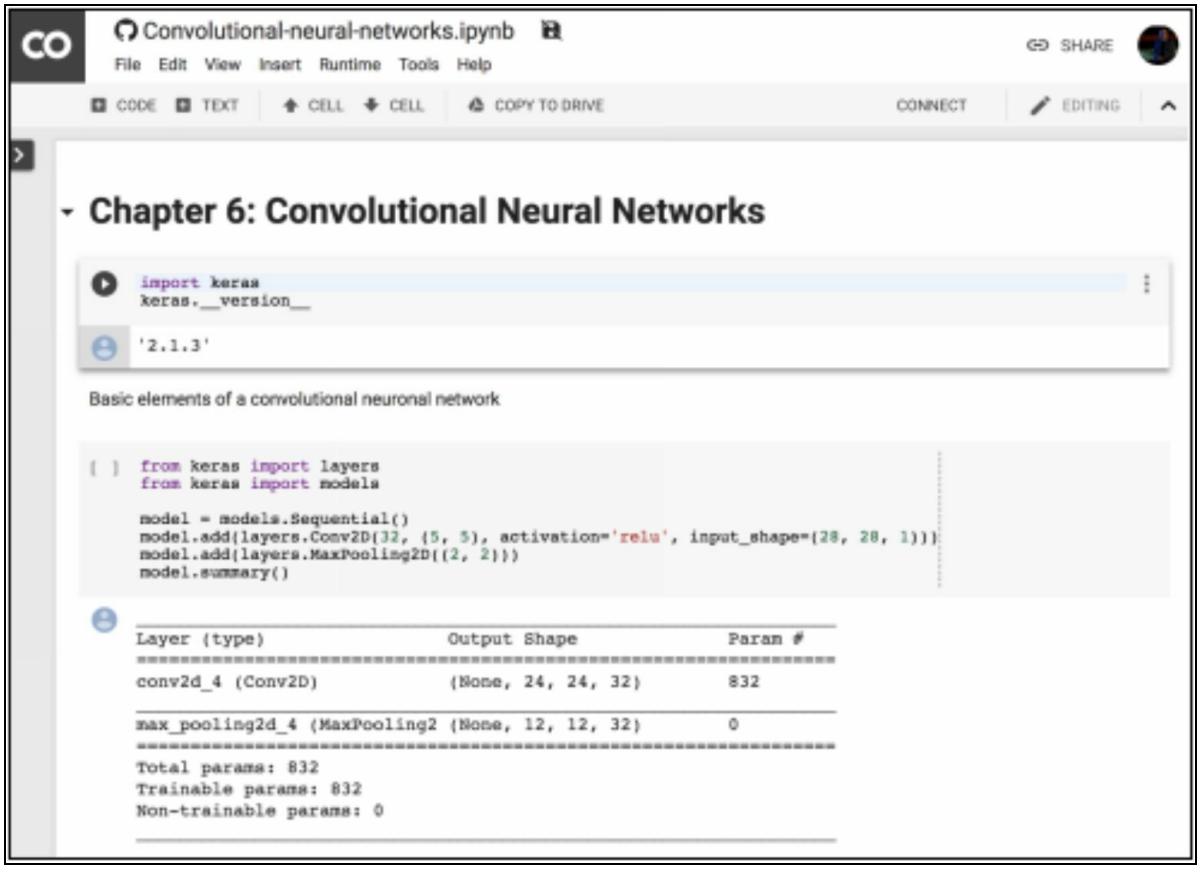
It is a Google research project created to help to disseminate Machine Learning education and research. It is a Jupyter notebook environment that requires no configuration and runs completely in the Cloud allowing the use of Keras, TensorFlow and PyTorch. The most important feature that distinguishes Colab from other free cloud services is; Colab provides GPU and is totally free. Detailed information about the service can be found on the faq page[7].

Notebooks are stored in Google Drive and can be shared as you would do with Google Docs. This environment is free to use, which only requires a Google account. In addition, the environment allows the use of an NVIDIA K80 GPU free of charge.

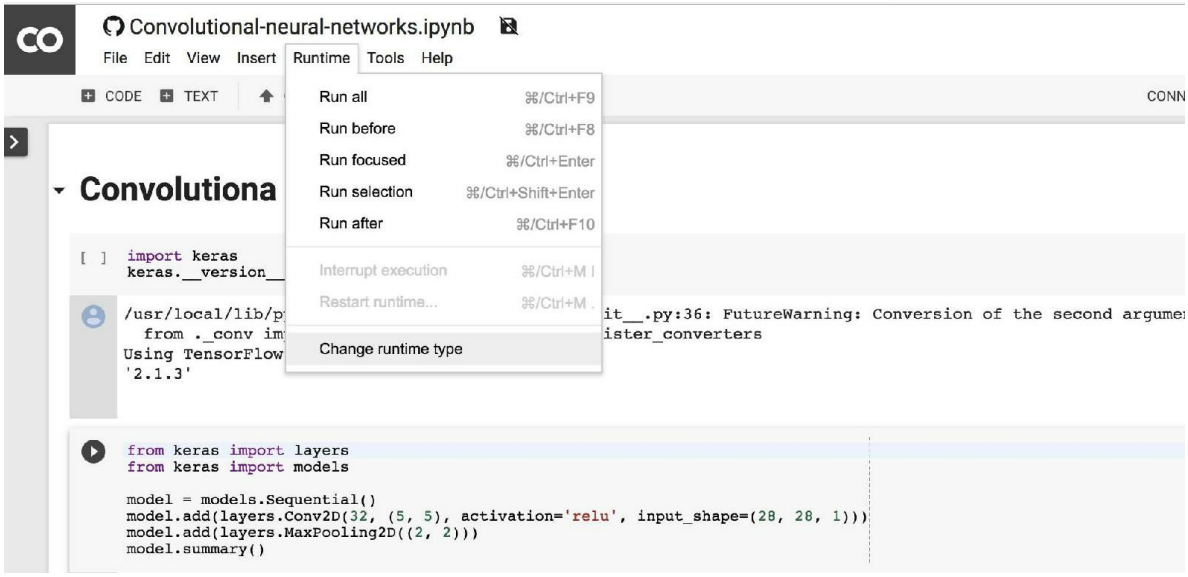
When entering for the first time you will see a window like the one shown below. In this window you should select the GITHUB tab and fill in the URL field with "JordiTorresBCN" and the Repository field with "jorditorresBCN / DEEP-LEARNING-practical-introduction-with-Keras".



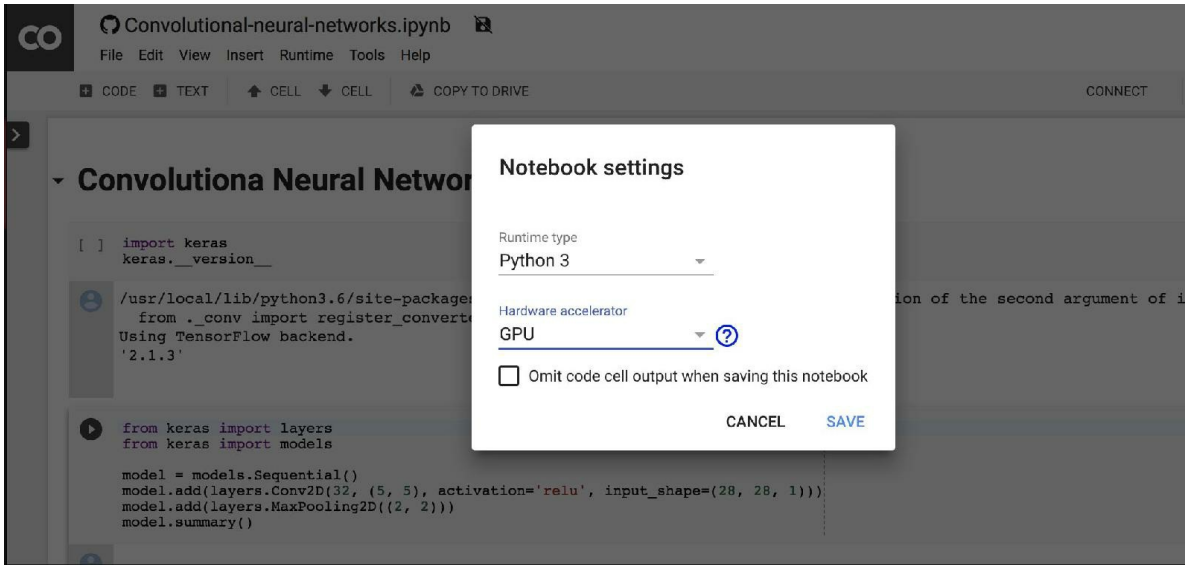
You will see three notebooks which will be the ones that we will use throughout the book. To load a notebook, click on the button that appears on their right (open notebook in new tab):



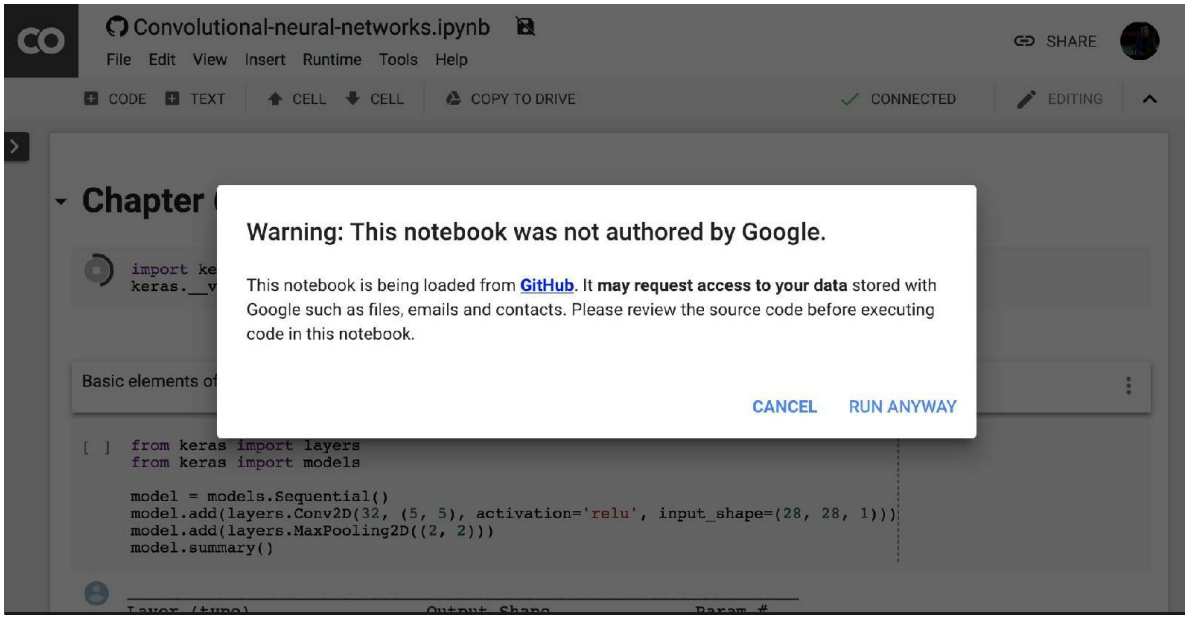
By default, Colab notebooks run on CPU. You can switch your notebook to run with GPU. In order to obtain access to one GPU we need to choose the tab Runtime and then select “Change runtime type” as shown in the following figure:



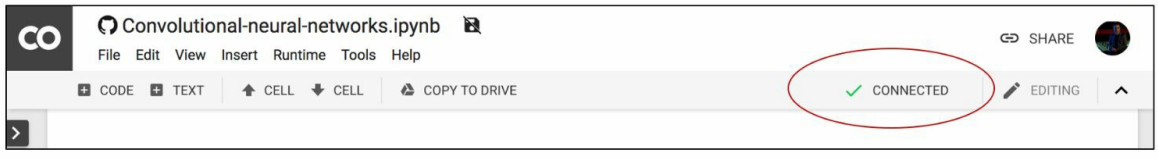
When a pop-up window appears select GPU. Ensure "Hardware accelerator" is set to GPU (the default is CPU).



A warning may appear indicating that the code is not created by Google. I hope that you trust my code and run it anyway! ;-)



Afterwards, ensure that you are connected to the runtime (there is a green check next to "connected" in the menu ribbon):



Now you are able to run GitHub repo in Google Colab. Enjoy!

Supercomputing, the heart of Deep Learning

Surely, at this point, some readers have already posed the question: why has a researcher in supercomputing such as me, started to investigate Deep Learning?

In fact, many years ago I started to be interested in how supercomputing could contribute to improving Machine Learning methods; Then, in 2006, I started co-directing PhD theses with a great friend, and professor at the Computer Science department of the UPC, Ricard Gavaldà[8] , an expert in Machine Learning and Data Mining.

But it was not until September 2013, when I already had a relatively solid base of knowledge about Machine Learning, that I started to focus my interest on Deep Learning. Thanks to the researcher from our Computer Architecture Department at UPC Jordi Nin, I discovered the article *Building High-level Features Using Large Scale Unsupervised Learning*[9], written by Google researchers. In this article presented at the previous International Conference in Machine Learning (ICML'12), the authors explained how they trained a Deep Learning model in a cluster of 1,000 machines with 16,000 cores. I was very happy to see how supercomputing made it possible to accelerate this type of applications, as I wrote in my blog[10] a few months later, justifying the reasons that led the group to add this research focus to our research roadmap.

Thanks to Moore's Law[11], in 2012, when these Google researchers wrote this article, we had supercomputers that allowed us to solve problems that would have been intractable a few years before due to the computing

capacity. For example, the computer that I had access to in 1982, where I executed my first program with punch-cards, it was a Fujitsu that made it possible to execute a little more than one million operations per second. 30 years later, in 2012, the Marenostrum supercomputer that we had at the time at the Barcelona Supercomputing Center-National Supercomputing Center (BSC), was only 1,000,000,000 times faster than the computer on which I started.

With the upgrade of that year, the MareNostrum supercomputer offered a theoretical maximum performance peak of 1.1 Petaflops (1,100,000,000,000 floating point operations per second[12]). It achieved it with 3,056 servers with a total of 48,896 cores and 115,000 Gigabytes of total main memory housed in 36 racks. At that time the Marenostrum supercomputer was considered to be one of the fastest in the world. It was placed in the thirty-sixth position, in the TOP500 list[13], which is updated every half year and ranks the 500 most powerful supercomputers in the world. Attached you can find a photography where you can see the Marenostrum computer racks that were housed in the

Torres Girona chapel of the UPC campus in Barcelona.[14].

The first GPU in the ImageNet competition

During that period was when I began to become aware of the applicability of supercomputing to this new area of research. When I started looking for research articles on the subject, I discovered the existence of the Imagenet competition and the results of the team of the University of Toronto in the competition in 2012[15]. The ImageNet competition (Large Scale Visual Recognition Challenge[16]) had been held since 2010, and by that time it had become a benchmark in the computer vision community for the recognition of objects on a large scale. In 2012 Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hilton used for the first time hardware accelerators GPU (graphical processing units) [17], which was already used at that time in supercomputing centers like ours in Barcelona to increase the speed of execution of applications that require the performance of many calculations.

For example, at that time BSC already had a supercomputer called MinoTauro, with 128 Bull505 nodes, equipped with 2 Intel processors and 2 Tesla M2090 GPUs from NVIDIA each one. With a peak performance of 186 Teraflops, launched in September 2011 (out of curiosity, at that time it was considered the most energy efficient supercomputer in Europe according to the Green500 list[18]).

Until 2012, the increase in computing capacity that we got each year from computers was as a result of the improvement of the CPU. However, since then the increase in computing capacity for Deep Learning has not only been credited to them, but also to the new massively parallel systems based on GPU accelerators, which are many times more efficient than traditional CPUs.

GPUs were originally developed to accelerate the 3D game that requires the

repeated use of mathematical processes that include different matrix calculations. Initially, companies such as NVIDIA and AMD developed these fast and massively parallel chips for graphics cards dedicated to video games. However, it soon became clear that the use of GPUs for 3D games was also very suitable for accelerating calculations on numerical matrices; therefore, this hardware actually benefited the scientific community, and in 2007 NVIDIA launched the CUDA[19] programming language to program its GPUs. As a result, supercomputing research centers such as the BSC began using GPU clusters to accelerate numerical applications.

But as we will see later in this book, artificial neural networks basically perform matrix operations that are also highly parallelizable. And this is what Alex Krizhevsky's team did in 2012: he trained his Deep Learning algorithm "AlexNet" with GPU. Since then, some research groups have started using GPUs for this competition, and nowadays all the groups that do research in Deep Learning research field are using this hardware or equivalent alternatives that have appeared recently.

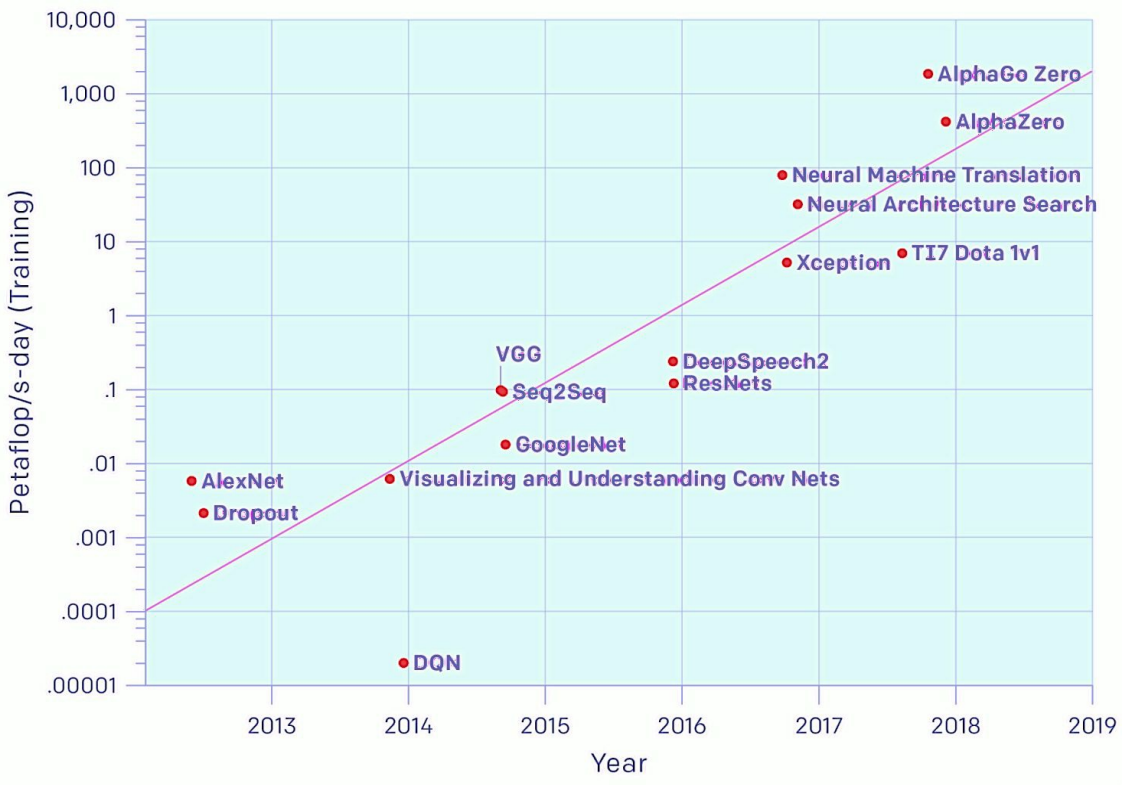
An exponential growth of computing capacity

I have already said that Krizhevsky's team's milestone was an important turning point in the field of Deep Learning, and since then there have been spectacular results, one after another, with an exponential growth of increasingly surprising results.

But I believe that research in this field has been guided largely by experimental findings rather than by theory, in the sense that these spectacular advances in the area since 2012 have only been possible thanks to the fact that the computation that was required to be able to carry them out was available; In this way, researchers in this field have been able to test and extend old ideas, while they have advanced with new ones that required a lot of computing resources.

OpenAI[20] has recently published a study on its blog[21] that corroborates precisely this vision that I am defending. Specifically, they present an analysis in which it is confirmed that, since 2012, the amount of computation available to generate models of artificial intelligence has increased exponentially while claiming that improvements in computing capacity have been a key component of the progress of Artificial Intelligence.

In this same article they present an impressive graph[22] to synthesize the results of their analysis:



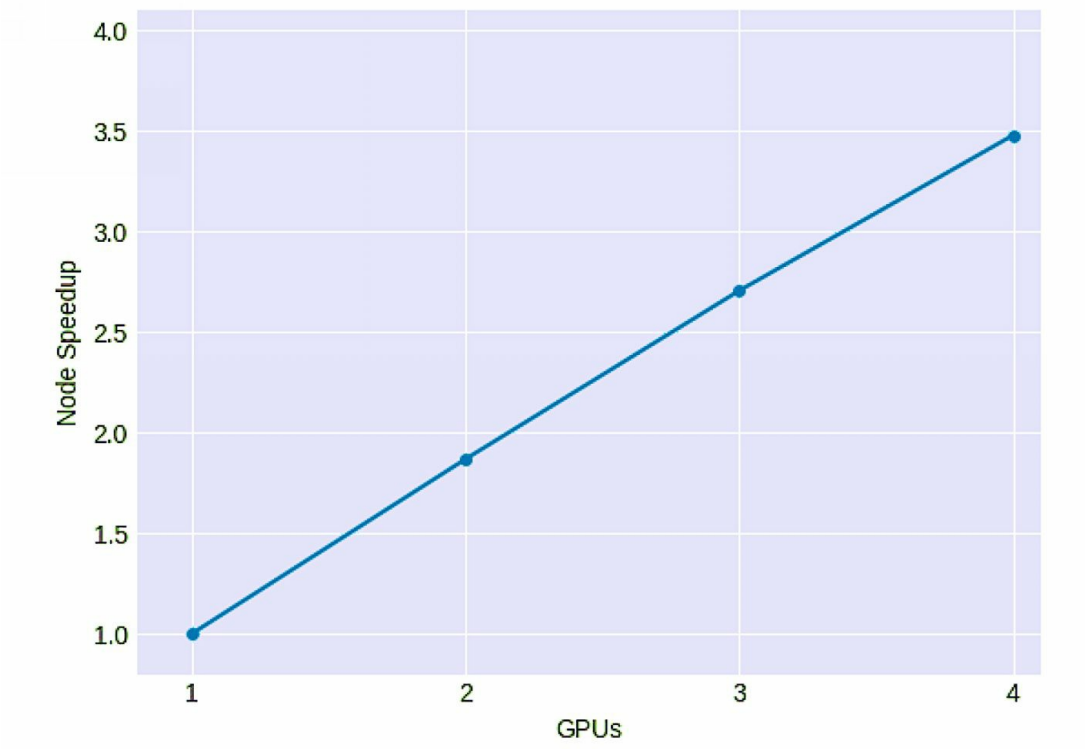
The graph shows the total amount of calculations, in Petaflop per day, that have been used to train neural networks that have their own name and are referents in the Deep Learning community. Remember that a "petaflop / s-day", the vertical axis of the graph that is in logarithmic scale, is equivalent to perform 1,000,000,000,000,000 neural network operations per second during a day (s-day), or a total of approximately 100,000,000,000,000,000

operations, regardless of numerical precision.

Accelerating Deep Learning with parallel systems

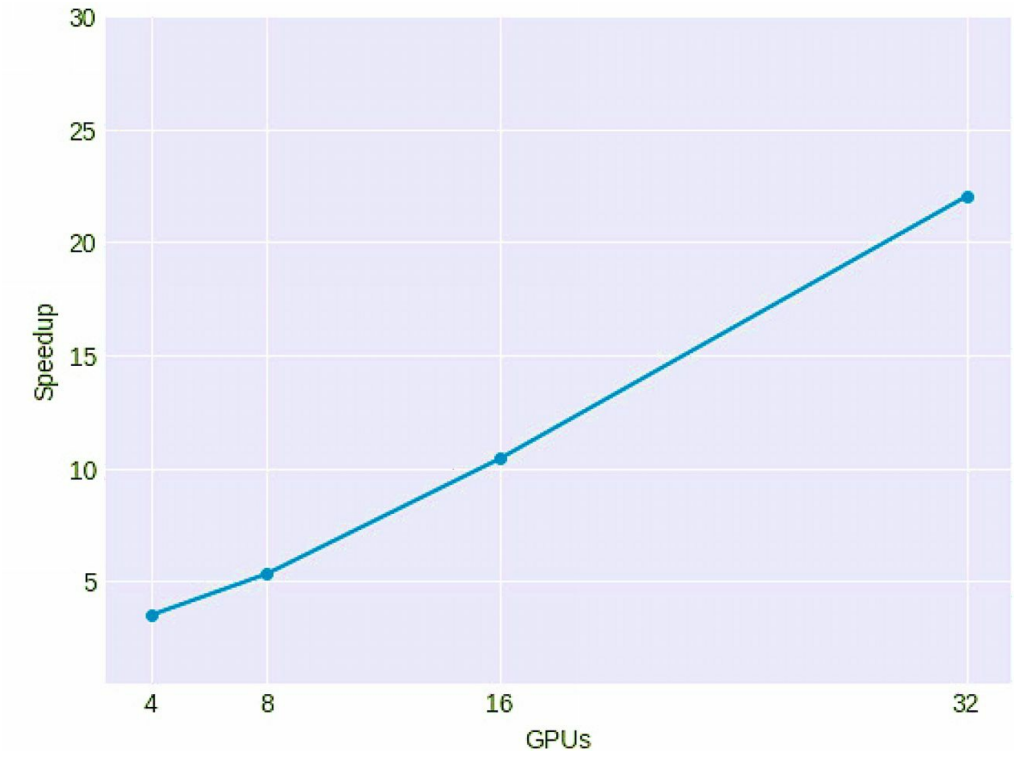
The tasks of training Deep Learning networks requires a large amount of computation and, often, they also need the same type of matrix operations as the numerical calculation intensive applications, which makes them similar to traditional supercomputing applications. Therefore, Deep Learning applications work very well in computer systems that use accelerators such as GPU or field-programmable gate arrays (FPGA), which have been used in the Supercomputing field for more than a decade within the walls of the supercomputing research centers. These hardware devices focus on computational performance by specializing their architecture in using high data parallelism in supercomputing workloads. And precisely these techniques can also be used to accelerate the learning algorithms of Deep Learning.

Therefore, from 2012 until 2014, Deep Learning researchers started using systems with GPUs. The advantage, in addition, is that these learning algorithms escalated perfectly when they could put more than one GPU in a node. The following graph, extracted from one of our research articles, shows how increasing the number of GPUs can accelerate the learning process[23]:



Accelerating Deep Learning with distributed systems

The large computational capacity available at that time allowed the Deep Learning community to move forward and be able to design increasingly complex neural networks, requiring more computing capacity than a server with multiple GPUs could offer. Therefore starting in 2014 to accelerate even more the calculation required, computing began to be distributed among multiple machines with several GPUs connected by a network. This solution had been adopted again previously, and very well known, in the community of researchers in supercomputing, specifically in the interconnection of machines through optical networks with low latency, which made it possible to do the distribution in a very efficient manner. The following graph shows how the same algorithm can be accelerated with several machines that each have 4 GPUs[24]:



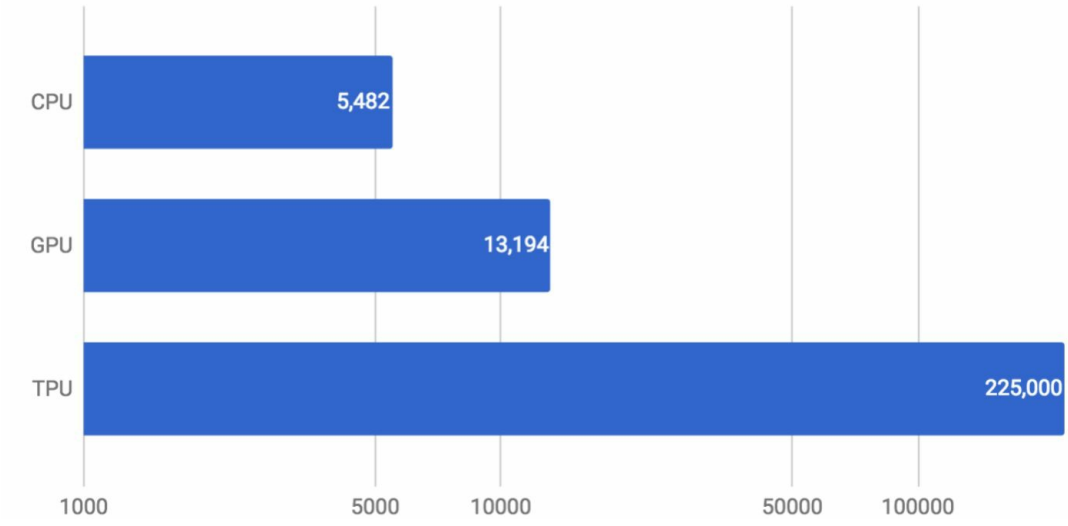
Also libraries of the standard of communication like Message Passing Interface (MPI) [25], used in the scientific community of supercomputing for decades, are spreading in distributed Deep Learning, requiring the knowledge of experts in supercomputing to accelerate these algorithms.

Will specialized hardware for deep learning be a game changer?

As of 2016, in addition to all the previous innovations in supercomputing, processing chips began to appear that were specially designed for Deep Learning algorithms. For example, in 2016 Google announced that it had built a dedicated processor called the Tensor Processing Unit (TPU) [26]. Since then Google has already developed 3 versions of TPU, the last one presented in its IO[27] conference, where they claimed that it is 8 times more powerful than the previous version. In addition, now not only the architecture is specific to train neural networks, but also for the inference stage (use of the

previously trained model).

In the following graph obtained from the Google Cloud blog[28], we can see a comparison of predictions per second obtained (on a logarithmic scale) for the three different types of architecture mentioned above.



The acceleration of Deep Learning with specialized hardware has only just begun, both for the training stage and the inference stage, if we take into account that numerous companies are appearing that are designing and starting to produce specific chips for Artificial Intelligence[29].

Specialized hardware is sure going to be a big factor in Deep Learning race. We will see great progress soon, I am sure. However, even more interesting will be to see what role Deep Learning plays in changing hardware in the near future.

Tapping the Next Generation of Supercomputers

And now we are at the point of convergence of Artificial Intelligence

technologies and supercomputing technologies. The result will soon be part of the portfolio that companies providing computer systems (and Cloud services) will offer to the industrial and business world.

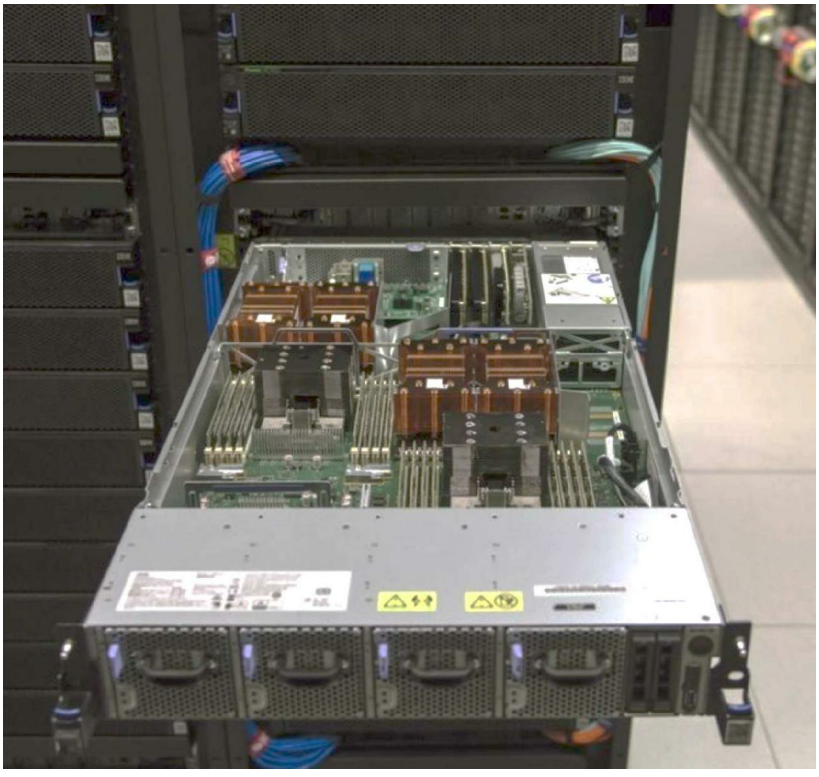
An example of what will be on the market in a short while, is a part of the current Supercomputer Marenostrum of the Barcelona Supercomputing Center (BSC). MareNostrum is the generic name used by the BSC to refer to the different updates of its most emblematic and the most powerful supercomputer in Spain, and until today four versions have been installed since 2004[30]. At present, the Marenustrum is the most heterogeneous supercomputer in the world, with all kinds of experimental hardware available on the market, since its purpose is to serve as an experimental platform to design future supercomputers.

This is structured in that the calculation capacity of the current MareNostrum 4 is divided into two parts of totally differentiated hardware: a block of general purpose and a block of emerging technologies. The emerging technologies block is made up of clusters of three different technologies that will be incorporated and updated as they become available. These are technologies that are currently being developed in the United States and Japan to accelerate the arrival of the new generation of pre-exascale supercomputers.

One of these technologies is based on an IBM system designed especially for Deep Learning and Artificial Intelligence applications[31]; IBM has created all the software stack necessary for it. At the time of writing this book, the hardware is available and the PowerAI[32] software package will soon be installed, which will convert this supercomputing hardware into a machine specially designed for Artificial Intelligence. Through this software, the main frameworks of Deep Learning such as TensorFlow (and Keras, included in the Tensorflow package), Caffe, Chainer, Torch and Theano will be available

to researchers of Artificial Intelligence.

In terms of hardware, this part of the Marenostrum consists of a 54 node cluster based on IBM POWER9 and NVIDIA V100 with Linux operating system and interconnected by an infiniband network at 100 Gigabits per second. Each node is equipped with 2 IBM POWER9 processors that have 20 physical cores each and 512GB of memory. Each of these POWER9 processors are connected to two NVIDIA V100 (Volta) GPUs with 16GB of memory, a total of 4 GPUs per node. See the following picture:



The new NVIDIA V100 GPUs are the most advanced GPUs yet[33] to accelerate Artificial Intelligence applications equivalent to 100 CPUs according to NVIDIA[34]. This is achieved by matching their CUDA cores with 640 core tensor, which did not have the previous family of GPU Pascal

from NVIDIA. The core tensor is specifically designed to multiply two matrices of 4×4 elements in floating point format and also allows the accumulation of a third matrix, thus being able to execute very quickly the basic operations of neural networks both in the inference phase and training phase.

In addition, this new version of GPU updates the bus with NVLINK 2.0[35] system that allows a high bandwidth with six links that can transfer 50 Gigabytes per second. Although traditionally the NVLINK bus was originally designed to connect the GPUs, this version also makes it possible to connect GPU and CPU. Another important element is access to memory, which has improved compared to previous versions and allows bandwidths of up to 900 GigaBytes per second. Something awesome!.

I describe all this detail so that it does not surprise you that with only 3 racks of the current Marenostrum[36] (the ones seen in the following picture) there is 1.5 Petaflops of theoretical maximum performance, much more than the 1.1 Petaflops that in 2012, the Marenostrum 3 had, in 36 racks (shown in the previous photo in page 30).

In summary, it is not my intention to give you a computer architecture class, but I would like to explain and make you aware, with real examples close to us, that the computing capacity is exponentially evolved. And it has allowed us , as I said before, to try new ideas or to extend the old ones, since many of the advances in the Deep Learning area since 2012 have been guided by the experimental findings with hardware equivalent to the one that was being used in the world of supercomputing.

However, without any doubt, other factors have contributed to trigger the

resurgence of Artificial Intelligence, and I must be fair and say that it is not only due to Supercomputing. If I asked you one of the other key factors, the Big Data phenomenon comes to mind as another of the facilitators of the resurgence of this new stage of Artificial Intelligence. But there are others that perhaps the reader is not so familiar with . In the next chapter we will present them.

A new disruptive technology is coming

Artificial intelligence is being considered as the new industrial revolution, heart of what some call industry 4.0. Well, Deep Learning is the engine of this process and in the following chapters we will talk about it extensively. But in this chapter we will situate the issue first, to see why artificial intelligence is already here and why it has come to stay.

Artificial Intelligence is changing our life

We are facing dizzying advances in the quality and device features of a wide range of everyday technologies: in the case of speech recognition, voice-to-text transcription has experienced incredible advances and is now available in different devices. We are interacting more and more with our computers (and all kinds of devices) by simply talking to them.

There have also been spectacular advances in natural language processing. For example, just by clicking on the *Google Translate*[37] micro icon (at the bottom left of the text box), the system transcribes into another language what is being dictated. *Google Translate* already allows you to convert sentences from one language to another in 32 language pairs, and offers text translation for more than 100.

In turn, the advances in computer vision are also enormous: now our computers, for example, can recognize images and generate textual descriptions of their content in seconds.

These three areas are crucial to unleash the improvements in robotics, drones or cars without driver, among many others areas. Artificial Intelligence is at the heart of all this technological innovation, which lately advances so quickly thanks to Deep Learning.

And all this despite the fact that artificial intelligence has not yet been widely deployed and it is difficult to get an idea of the great impact it will have, just as in 1995 it was difficult to imagine the future impact of the internet. Back then, most people did not see how the internet would end up being as relevant to them and how it was going to change their lives.

It is said that the first industrial revolution used steam energy to mechanize production in the second half of the 18th century; the second revolution used electricity to boost mass production in the mid-nineteenth century while, in the third one, electronics and software were used in the seventies of the last

century. Today we are facing a new source of value creation in the area of information processing where everything will change. In different forums, people already talk about a fourth industrial revolution, marked by technological advances such as Artificial Intelligence (Machine Learning, Deep Learning, etc.) and in which computers will be "even wiser".

2 Artificial Intelligence, Machine Learning and Deep Learning

Before continuing we will specify a bit what we understand by Artificial Intelligence, Machine Learning and Deep Learning, three terms that will appear very often throughout the book.

Artificial Intelligence

What do we mean when we talk about Artificial Intelligence? An extensive and precise definition (and description of its areas) is found in the book *Artificial Intelligence, a modern approach*[38] written by Stuart Rusell[39] and Peter Norvig[40], the most popular book on Artificial Intelligence in the university world and, without a doubt for me, the best starting point to have a global vision of the subject. But trying to make a more general approach (purpose of this book), we could accept a simple definition in which by Artificial Intelligence we refer to that intelligence shown by machines, in contrast to the natural intelligence of humans. In this sense, a possible concise and general definition of Artificial Intelligence could be the effort to automate intellectual tasks normally performed by humans.

As such, the area of artificial intelligence is a very broad scientific field that covers many areas of knowledge related to machine learning; even many more approaches not always cataloged as Machine Learning are included by my university colleagues who are experts in the subject. Besides, over time, as computers have been increasingly able to "do things", tasks or technologies considered as "smart" have been changing.

Furthermore, since the 1950s, Artificial Intelligence has experienced several waves of optimism, followed by disappointment and loss of funding and interest (periods known as AI winter[41]), followed by new approaches,

success and financing. Moreover, during most of its history, Artificial Intelligence research has been dynamically divided into subfields based on technical considerations or concrete mathematical tools and with research communities that sometimes did not communicate sufficiently with each other.

Machine Learning

As we said in the previous section, advances such as speech recognition, natural language processing or computer vision are crucial to trigger improvements in robotics, drones, driverless cars, among many other areas that are changing the near future. Many of these advances have been possible thanks to a family of techniques popularly known as Deep Learning, of which we will talk extensively. But first, for us to have a correct global image, I think it's interesting to specify that Deep Learning is a subpart of one of the areas of Artificial Intelligence known as Machine Learning.

Machine Learning, is in itself a large field of research and development. In particular, Machine Learning could be defined as the subfield of Artificial Intelligence that gives computers the ability to learn without being explicitly programmed, that is, without requiring the programmer to indicate the rules that must be followed to achieve their task; the computers do them automatically.

Generalizing, we can say that Machine Learning consists in developing for each problem a prediction “algorithm" for a particular use case. These algorithms learn from the data in order to find patterns or trends to understand what *the data tell* us and in this way build a model to predict and classify the elements.

Given the maturity of the research area in Machine Learning, there are many well-established approaches to Machine Learning. Each of them uses a different algorithmic structure to optimize the predictions based on the received data. Machine Learning is a broad field with a complex taxonomy of

algorithms that are grouped, in general, into three main categories: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

We mean that learning is supervised when the data we use for training includes the desired solution, called "label". Some of the most popular machine learning algorithms in this category are linear regression, logistic regression, support vector machines, decision trees, random forest or neural networks.

On the other hand, when the training data do not include the labels, we refer to an Unsupervised Learning and it will be the algorithm which will try to classify the information by itself. Some of the best-known algorithms in this category are clustering (K-means) or principal component analysis (PCA).

We also talk about Reinforcement Learning when the model is implemented in the form of an agent that should explore an unknown space and determine the actions to be carried out through trial and error: the algorithm will learn by itself thanks to the rewards and penalties that it obtains from its “actions”. The agent must create the best possible strategy (policies) to obtain the greatest reward in time and form. This learning allows it to be combined with other types, and is now very fashionable since the real world presents many of these scenarios.

Basic Machine Learning terminology

In this section we will advance basic Machine Learning terminology that will allow us to keep a presentation script of the concepts of Deep Learning in a more comfortable and gradual way throughout the book.

In Machine Learning we refer to *label* to what we are trying to predict with a model. Instead, we call an input variable a *feature*.

A *model* defines the relationship between features and labels and has two clearly differentiated phases for the subject that concerns us:

- *Training* phase (or learning plase), which is when the model is created or learned, showing the examples of input that have been tagged; In this way, the model is able to iteratively learn the relationships between the features and labels of the examples.
- *Inference* phase (or prediction or phase), which refers to the process of making predictions by applying the model already trained to non-labeled examples.

Consider a simple example of a model that expresses a linear relationship between features and labels. The model could be expressed as follows:

$$y=wx+b$$

Where:

- ***y*** is the *label* of an input example.
- ***x*** is the *feature* of that input example.
- ***w*** is the slope of the line and that is what in general we call *weight*. It is one of the two parameters that the model has to learn during the training process to be able to use it later for inference.
- ***b*** is the intersection point of the line on the y-axis that we call *bias*. This is the other parameter that must be learned by the model.

Although in this simple model that we have represented there is only one input feature, in the case of Deep Learning we will see that we have many input variables, each with its w_i weight. For example, a model based on three features (x_1, x_2, x_3) can be expressed as follows:

$$y = w_1x_1 + w_2x_2 + w_3x_3 + b$$

Or, more generally, it can be expressed as:

$$y = \sum_i w_i x_i + b$$

which expresses the sum of the scalar product between the two vectors (X and W) and then adds the bias. The parameter bias *b*, in order to facilitate the formulation, it is sometimes expressed as the parameter w_0 (assuming a fixed additional entry of x_0=1).

In the training phase of a model, ideal values are learned for the model parameters (the w_i weights and the *b* bias). In supervised learning, the way to achieve this is to apply an automatic learning algorithm that obtains the value of these parameters by examining many labeled examples and try to determine values for these model parameters that minimize what we call *loss*.

As we will see throughout the book, the loss is a central concept in Deep Learning that represents the penalty of a bad prediction. That is, the loss is a number that indicates how bad a prediction has been in a particular example (if the prediction of the model is perfect, the loss is zero). To determine this value, as we will see later, in the training process the concept of the loss function will appear, and for the time being we can see how the mathematical function that aggregates the individual losses obtained from the input examples to the model.

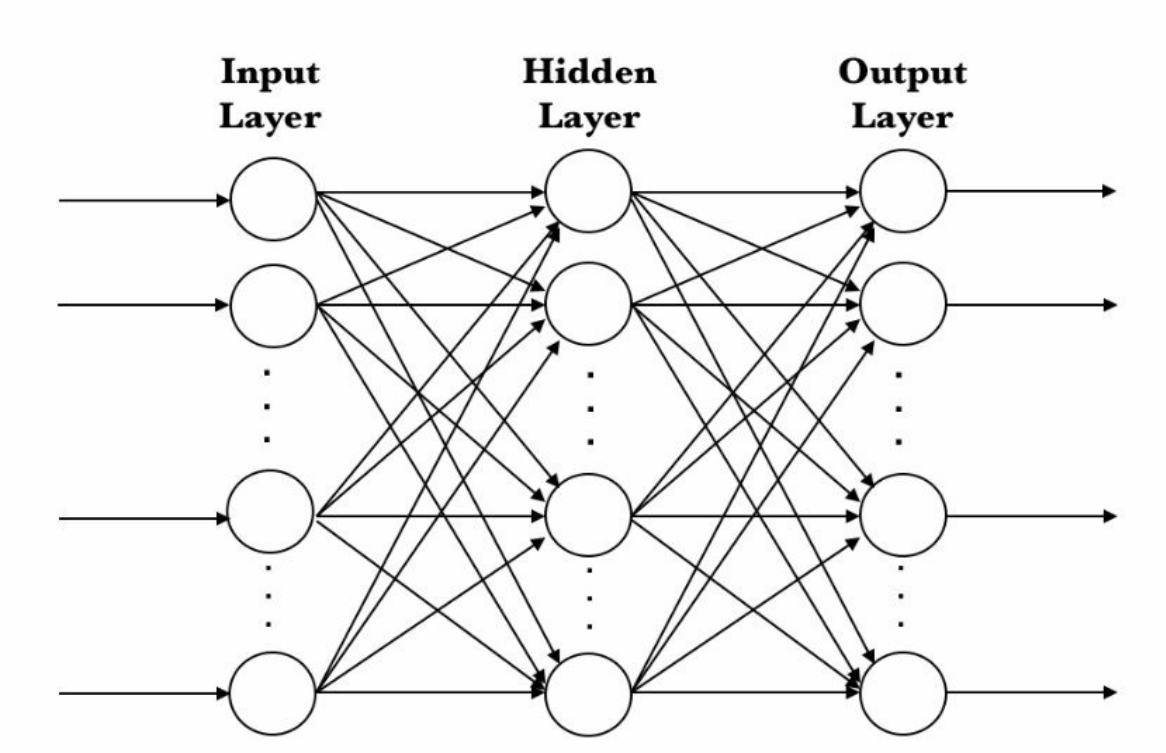
In this context, for now we can consider that the training phase of a model consists basically in adjusting the parameters (the weights w_i and the *b* bias) in such a way that the result of the loss function returns the minimum possible value.

Finally, we need to address the concept of overfitting in a model, which occurs when the model obtained is adjusted so much to the labeled examples of input that, later, the resulting model cannot make the correct predictions with new data examples that have never been seen before. Given the introductory nature of this book we will not go into this topic, but it is a central theme in Deep Learning.

Artificial neural networks and Deep Learning

A special case of Machine Learning algorithms are artificial neural networks, whose algorithmic structures allow models that are composed of multiple layers of processing to learn data representations with multiple levels of abstraction. This set of layers composed of "neurons" performs a series of linear and non-linear transformations to the input data to generate an output close to the expected (label). Supervised learning, in this case, consists in obtaining the parameters of these transformations (the w_i weights and the b bias) and attempting that these transformations produce an output that differ as little as possible to the expected output.

A simple graphical approach to a Deep Learning neural network can be



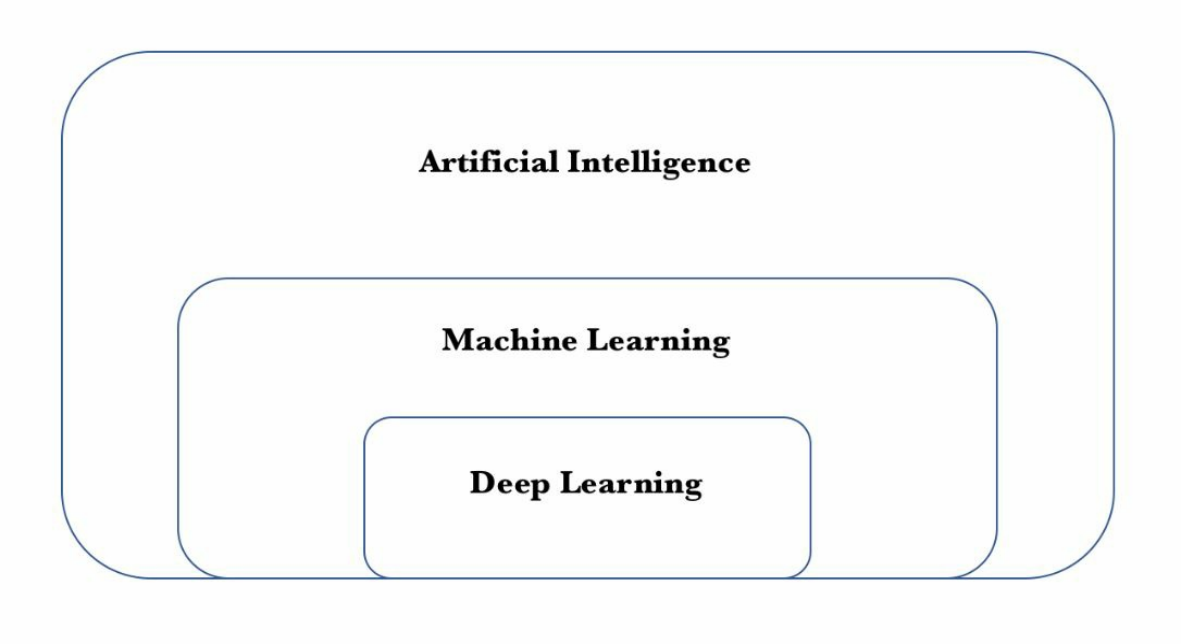
Specifically, in this image we represent an artificial neural network with 3 layers: an input layer that receives the input data and an output layer that returns the prediction made. The layers that we have in the middle are called hidden layers and we can have many, each one with a different number of neurons. We will see later that neurons, represented by the circles, will be interconnected with each other in different ways between the neurons of the different layers.

In general, today we are handling artificial neural networks with many layers, which are literally stacked one on top of the other; hence the concept of deep (depth of the network). Each of them is composed of many neurons, each with its parameters (the weights w_i and the bias b) which perform a simple transformation of the data that they receive from neurons of the previous layer to pass them to those of the posterior layer. The union of all these

transformations carried out by the neurons of the network is what allows discovering complex patterns in the data.

Before finishing this section, I would like to give the reader a magnitude of the problem that involves programming the Deep Learning algorithms these days: different layers serve different purposes, and each parameter and hyperparameter matter a lot in the final result; this makes it extremely complicated when trying to refine the programming of a neural network model, looking more like an art than a science for those who enter the area for the first time. But this does not imply that it is something mysterious, although it is true that much remains to be investigated, but it simply takes many hours of learning and practice.

The next figure visually summarizes the intuitive idea that Deep Learning is just a part of Artificial Intelligence, although nowadays it is probably the most dynamic one and the one that is captivating the scientific community. And the same way I previously mentioned the work of Stuart Rusell and Peter Novig as the main book on Artificial Intelligence, when it comes to Deep Learning we can read an excellent book titled *Deep Learning*[42], by Ian Goodfellow, Yoshua Bengio and Aaron Corville, wchich constitutes the grounds for learning about this topic more deeply.


Artificial Intelligence
Machine Learning
Deep Learning

3 Why now?

In just ten years[43], four of the five largest companies in the world by market capitalization have changed: Exxon Mobil, General Electric, Citigroup and Shell Oil are out and Apple, Alphabet (the parent company of Google), Amazon and Facebook have taken their place. Only Microsoft maintains its position. You have already realized that all of them dominate the new digital era in which we find ourselves immersed. We are talking about companies that base their power on Artificial Intelligence in general and Deep Learning in particular.

John McCarthy coined the term Artificial Intelligence in the 1950s, being one of the founding fathers of Artificial Intelligence along with Marvin Minsky. Also in 1958 Frank Rosenblatt built a prototype neuronal network, which he called the *Perceptron*. In addition, the key ideas of the Deep Learning neural networks for computer vision were already known in 1989; also the fundamental algorithms of Deep Learning for time series such as LSTM were already developed in 1997, to give some examples. So, why now this Artificial Intelligence boom?

Undoubtedly, the available computing has been the main trigger, as we have already presented previously. However, other factors have contributed to unleash the potential of Artificial Intelligence and related technologies. Next we are going to talk about the most important factors that have influenced it.

The data, the fuel for Artificial Intelligence

Artificial Intelligence requires large datasets for the training of its models but, fortunately, the creation and availability of data has grown exponentially thanks to the enormous decrease in cost and increased reliability of data generation: digital photos, cheaper and precise sensors, etc. Furthermore, the

improvements in storage hardware of recent years, associated with the spectacular advances in technology for its management with NoSQL[44] databases, have allowed having enormous datasets to train Artificial Intelligence models.

Beyond the increases in the availability of data that the advent of the Internet has led to recently, specialized data resources have catalyzed the progress of the area. Many open databases have supported the rapid development of Artificial Intelligence algorithms. An example is the ImageNet[45] database, of which we have already spoken, freely available with more than 10 million images tagged by hand. But what makes ImageNet special is not precisely its size, but the competition that was carried out annually with it, being an excellent way to motivate researchers and engineers.

While in the early years the proposals were based on traditional computer vision algorithms, in 2012 Alex Krizhevsky used a Deep Learning neural network, now known as *AlexNet*, which reduced the error rate to less than half of what the winner of the previous edition of the competition got. Already in 2015, the winning algorithm rivaled human capabilities, and today Deep Learning algorithms far exceed the error rates in this competition of those who have humans.

But ImageNet is only one of the available databases that have been used to train Deep Learning networks lately; many others have been popular, such as: MNIST[46], STL[47], COCO[48], Open Images[49], Visual Question Answering[50], SVHN[51], CIFAR-10/100[52], Fashion-MNIST[53], IMDB Reviews[54], Twenty Newsgroups[55], Reuters-21578[56], WordNet[57], Yelp Reviews[58], Wikipedia Corpus[59], Blog Authorship Corpus[60], Machine Translation of European Languages[61], Free Spoken Digit Dataset[62], Free Music Archive[63], Ballroom[64], The Million Song[65], LibriSpeech[66],

VoxCeleb[67], The Boston Housing[68], Pascal[69] , CVPPP Plant Leaf Segmentation[70], Cityscapes[71].

It is also important to mention here Kaggle[72], a platform that hosts competitions of data analysis where companies and researchers contribute and share their data while data engineers from around the world compete to create the best prediction or classification models.

Entering into an era of computation democratization

However, what happens if you do not have this computing capacity in your company? Artificial Intelligence has until now been mainly the toy of big technology companies like Amazon, Baidu, Google or Microsoft, as well as some new companies that had these capabilities. For many other businesses and parts of the economy, artificial intelligence systems have so far been too expensive and too difficult to fully implement the hardware and software required.

But now we are entering another era of democratization of computing, and companies can have access to large data processing centers of more than 28,000 square meters (four times the field of Barcelona football club (Barça)), with hundreds of thousands of servers inside. We are talking about Cloud Computing[73].

Cloud Computing has revolutionized the industry through the democratization of computing and has completely changed the way business operates. And now it is time to change the scenario of Artificial Intelligence and Deep Learning, offering a great opportunity for small and medium enterprises that cannot build this type of infrastructure, although Cloud Computing can offer it to them; in fact, it offers access to a computing capacity that previously was only available to large organizations or

governments.

Besides, Cloud providers are now offering what is known as Artificial Intelligence algorithms as a Service (AI-as-a-Service), Artificial Intelligence services through Cloud that can be intertwined and work together with internal applications of companies through simple protocols based on API REST[74].

This implies that it is available to almost everyone, since it is a service that is only paid for the time used. This is disruptive, because right now it allows software developers to use and put virtually any artificial intelligence algorithm into production in a heartbeat.

Amazon, Microsoft, Google and IBM are leading this wave of AIaaS services, are put into production quickly from the initial stages (training). At the time of writing this book, Amazon AIaaS was available at two levels: predictive analytics with *Amazon Machine Learning*[75] and the *SageMaker*[76] tool for rapid model building and deployment. Microsoft offers its services through its *Azure Machine Learning* which can be divided into two main categories as well: *Azure Machine Learning* Studio[77] and *Azure Intelligence Gallery*[78]. Google offers *Prediction API*[79] and the Google *ML Engine*[80]. IBM offers AIaaS services through its *Watson Analytics*[81]. And let's not forget solutions that already come from *startups*, like PredicSis[82] or BigML[83].

Undoubtedly, Artificial Intelligence will lead the next revolution. Its success will depend to a large extent on the creativity of the companies and not so much on the hardware technology, in part thanks to Cloud Computing.

An open-source world for the Deep Learning community

Some years ago, Deep Learning required experience in languages such as C++ and CUDA; Nowadays, basic Python skills are enough. This has been possible thanks to the large number of open source software frameworks that have been appearing, such as Keras, central to our book. These frameworks greatly facilitate the creation and training of the models and allow abstracting the peculiarities of the hardware to the algorithm designer to accelerate the training processes.

The most popular at the moment are TensorFlow, Keras and PyTorch, because they are the most dynamic at this time if we rely on the contributors and commits or stars of these projects on GitHub[84].

In particular, TensorFlow has recently taken a lot of impulse and is undoubtedly the dominant one. It was originally developed by researchers and engineers from the Google Brain group at Google. The system was designed to facilitate Machine Learning research and make the transition from a research prototype to a production system faster. If we look at the Gihub page of the project[85] we will see that they have, at the time of writing this book, more than 35,000 commits, more than 1500 contributors and more than 100,000 stars. Not despicable at all.

TensorFlow is followed by Keras[86], a high level API for neural networks, which makes it the perfect environment to get started on the subject. The code is specified in Python, and at the moment it is able to run on top of three outstanding environments: TensorFlow, CNTK or Theano. Keras has more than 4500 commits, more than 700 contributors and more than 30,000 stars[87].

PyTorch and Torch[88] are two Machine Learning environments implemented in C, using OpenMP[89] and CUDA to take advantage of highly parallel infrastructures. PyTorch is the most focused version for Deep Learning and based on Python, developed by Facebook. It is a popular environment in this

field of research since it allows a lot of flexibility in the construction of neural networks and has dynamic tensors, among other things. At the time of writing this book, Pytorch has more than 12,000 commits, around 650 contributors and more than 17,000 stars[90].

Finally, and although it is not an exclusive environment of Deep Learning, it is important to mention Scikit-learn[91], that is used very often in the Deep Learning community for the preprocessing of data[92]. Scikit-learn has more than 22500 commits, more than 1000 contributors and nearly 30,000 stars[93].

But as we have already advanced, there are many other frameworks oriented to Deep Learning. Those that we would highlight are Theano[94] (Montreal Institute of Learning Algorithms), Caffe[95] (University de Berkeley), Caffe2[96] (Facebook Research) , CNTK[97] (Microsoft), MXNET[98] (supported by Amazon among others), Deeplearning4j[99], Chainer[100] , DIGITS[101] (Nvidia), Kaldi[102], Lasagne[103], Leaf[104], MatConvNet[105], OpenDeep[106], Minerva[107] and SoooA[108] , among many others.

An open-publication ethic

In the last few years, in this area of research, in contrast to other scientific fields, a culture of open publication has been generated, in which many researchers publish their results immediately (without waiting for the approval of the peer review usual in conferences) in databases such as arxiv.org of Cornell University (arXiv)[109]. This implies that there are numerous softwares available in open source associated with these articles, which allow this field of research to move tremendously quickly, since any new discovery is immediately available for the whole community to see it and, if it is the case, build on top a new proposal.

This is a great opportunity for users of these techniques. The reasons for research groups to openly publishing their latest advances can be diverse. For example, articles rejected in main conferences in the area can propagate solely as a preprint on arxiv. This is the case of one key paper for the advancement of Deep Learning written by G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. Salakhutdinov that introduced Dropout mechanism[110]. This paper was rejected from NIPS in 2012[111].

Or Google, when publishing the results, consolidates its reputation as a leader in the sector, attracting the next wave of talent, which is one of the main obstacles to the advancement of the topic.

Advances in algorithms

Thanks to the improvement of the hardware that we have already presented and to having more computing capacity by the scientists who were researching in the area, it has been possible to advance dramatically in the design of new algorithms that have allowed to overcome important limitations detected in the previous algorithms. For example, not until many years ago it was very difficult to train multilayer networks from an algorithm point of view. But in this last decade there have been impressive advances with improvements in activation functions, use of pre-trained networks, improvements in training optimization algorithms, etc. Today, algorithmically speaking, we can train models of hundreds of layers without any problem.

Densely Connected Networks

In the same way that when you start programming in a new language there is a tradition of doing it with a *Hello World print*, in Deep Learning you start by creating a recognition model of handwritten numbers. Through this example, this chapter will present some basic concepts of neural networks, reducing theoretical concepts as much as possible, with the aim of offering the reader a global view of a specific case to facilitate the reading of the subsequent chapters where different topics in the area will be dealt with in more detail.

Case of study: digit recognition

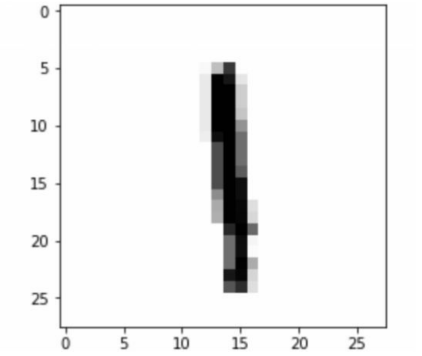
In this section we introduce the data we will use for our first example of neural networks: the MNIST dataset, which contains images of handwritten digits.

The MNIST dataset, which can be downloaded from *The MNIST database* page[112], is made up of images of hand-made digits. This dataset contains 60,000 elements to train the model and 10,000 additional elements to test it, and it is ideal for entering pattern recognition techniques for the first time without having to spend much time preprocessing and formatting data, both very important and expensive steps in the analysis of data and of special complexity when working with images; this dataset only requires small changes that we will comment on below.

This dataset of black and white images has been normalized to 20×20 pixels while retaining their aspect ratio. In this case, it is important to note that the images contain gray levels as a result of the *anti-aliasing*[113] technique, used in the normalization algorithm (reducing the resolution of all images to one of lower). Subsequently, the images were centered on a 28×28 pixel, calculating the center of mass of these and moving the image in order to position this point in the center of the 28×28 field. The images are of the following style:

Furthermore, the dataset has a label for each of the images that indicates what digit it represents, being therefore a supervised learning which we will discuss in this chapter.

This input image is represented in a matrix with the intensities of each of the 28×28 pixels with values between [0, 255]. For example, this image (the eighth of the training set):



It is represented with this matrix of 28×28 points (the reader can check it in the notebook of this chapter):

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   5  63 197   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 230  24   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254  48   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 255  48   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254  57   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254 108   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  16 239 254 143   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 143   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 143   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 162   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 240   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 113 254 240   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  83 254 245  31   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  79 254 246  38   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 214 254 150   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 241   8   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 240   2   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 254  82   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 230 247  40   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 168 209  31   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
```

```
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

On the other hand, remember that we have the labels, which in our case are numbers between 0 and 9 that indicate which digit the image represents, that is, to which class it is associated. In this example, we will represent each label with a vector of 10 positions, where the position corresponding to the digit that represents the image contains a 1 and the rest contains 0s. This process of transforming the labels into a vector of as many zeros as the number of different labels, and putting a 1 in the index corresponding to the label, is known as *one-hot encoding*.

2 *Perceptron*

Before moving forward, a brief intuitive explanation of how a single neuron works to fulfill its purpose of learning from the training dataset can be helpful for the reader. Let's look at a very simple example to illustrate how an artificial neuron can learn.

Regression algorithms

Based on what has been explained in the previous chapter, let us make a brief reminder about classic Machine Learning regression and classification algorithms since they are the starting point of our Deep Learning explanations.

We could say that regression algorithms model the relationship between different input variables (*features*) using a measure of error, the *loss*, which will be minimized in an iterative process in order to make predictions "as accurate as possible". We will talk about two types: logistic regression and linear regression.

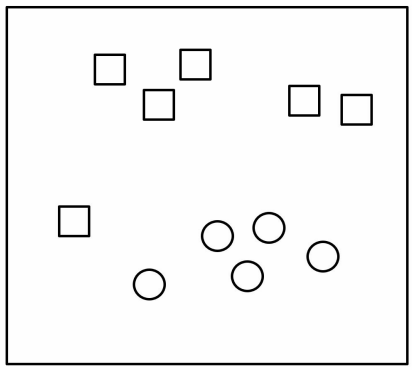
The main difference between logistic and linear regression is in the type of output of the models; when our output is discrete, we talk about logistic regression, and when the output is continuous we talk about linear regression.

Following the definitions introduced in the first chapter, logistic regression is an algorithm with supervised learning and is used to classify. The example that we will use next, which consists of identifying which class each input example belongs to by assigning a discrete value of type 0 or 1, is a binary classification.

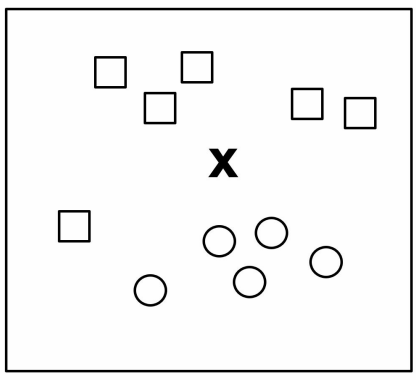
A plain artificial neuron

In order to show how a basic neuronal is, let's suppose a simple example

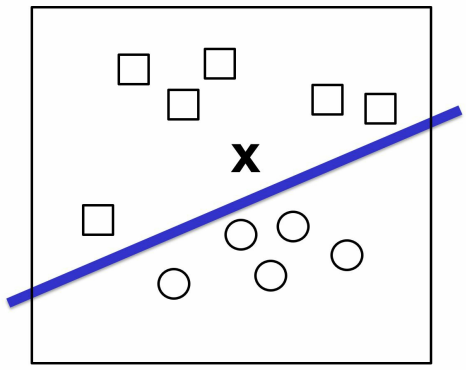
where we have a set of points in a two-dimensional plane and each point is already labeled "square" or "circle":



Given a new point "*X*", we want to know what label corresponds to it:



A common approach is to draw a line that separates the two groups and use this line as a classifier:



In this case, the input data will be represented by vectors of the form (x_1, x_2) that indicate their coordinates in this two-dimensional space, and our function will return '0' or '1' (above or below the line) to know if it should be classified as "square" or "circle". As we have seen, it is a case of linear regression, where "the line" (the classifier) following the notation presented in chapter 1 can be defined by:

$$y = w_1x_1 + w_2x_2 + b$$

More generally, we can express the line as:

$$y = W * X + b$$

To classify input elements X, which in our case are two-dimensional, we must learn a vector of weight W of the same dimension as the input vectors, that is, the vector (w_1, w_2) and a b bias.

With these calculated values, we can now construct an artificial neuron to classify a new element X. Basically, the neuron applies this vector W of calculated weights on the values in each dimension of the input element X,

and at the end adds the bias *b*. The result of this will be passed through a non-linear "activation" function to produce a result of '0' or '1'. The function of this artificial neuron that we have just defined can be expressed in a more formal way such as:

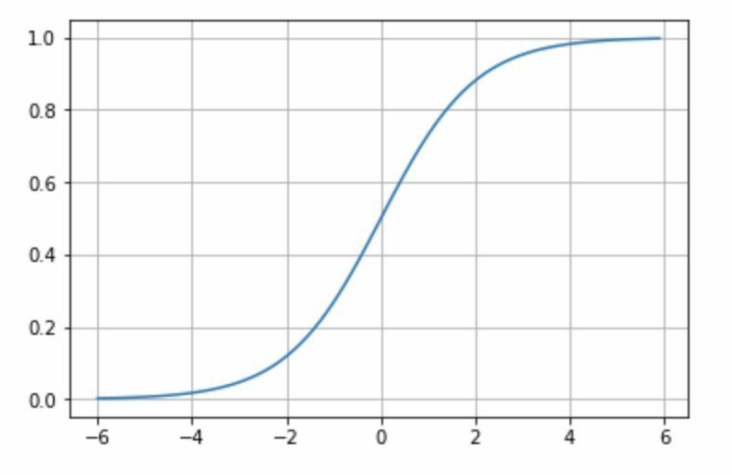
$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 \text{ si } z \geq 0 \\ 0 \text{ si } z < 0 \end{cases}$$

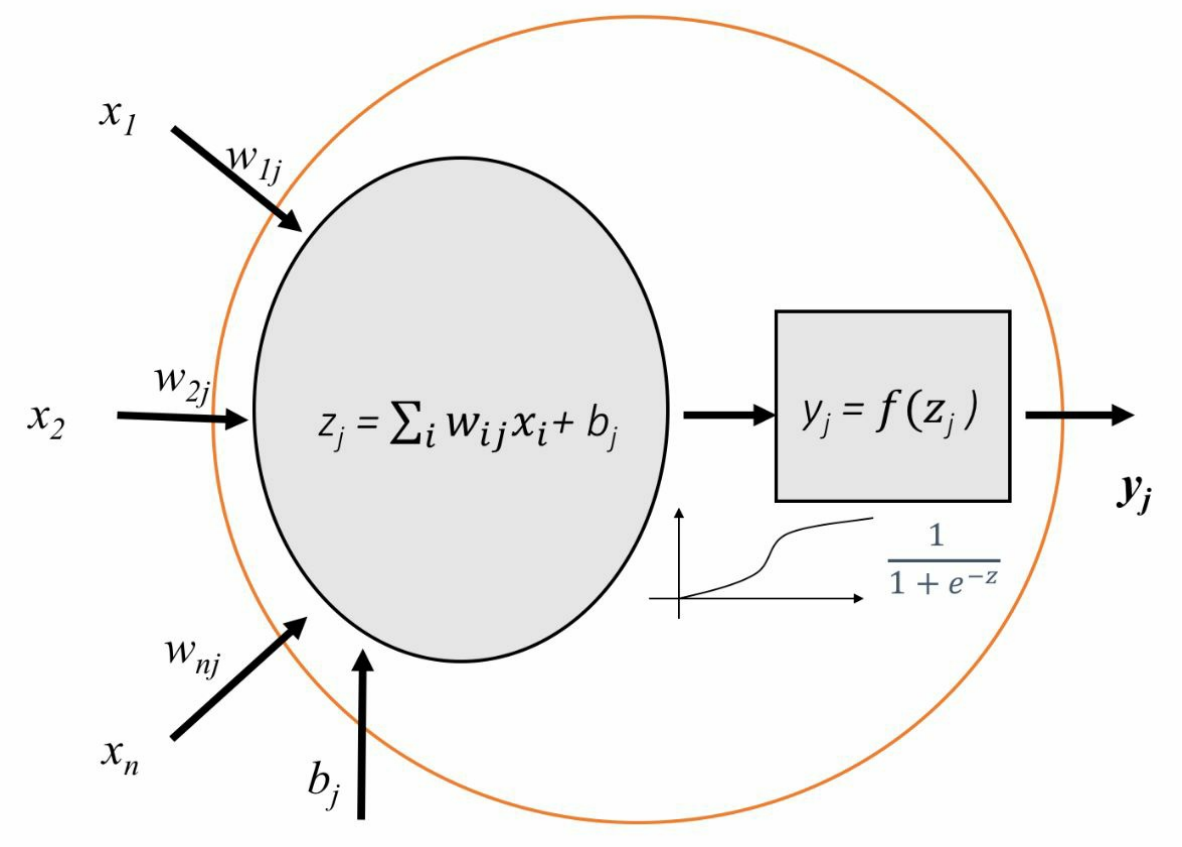
Now, we will need a function that applies a transformation to variable *z* so that it becomes '0' or '1'. Although there are several functions (which we will call "activation functions" as we will see in chapter 4), for this example we will use one known as a *sigmoid*[114] function that returns an actual output value between 0 and 1 for any input value:

$$y = \frac{1}{1+e^{-z}}$$

If we analyze the previous formula, we can see that it always tends to give values close to 0 or 1. If the input z is reasonably large and positive, "e" at minus *z* is zero and, therefore, the *y* takes the value of 1. If *z* has a large and negative value, it turns out that for "e" raised to a large positive number, the denominator of the formula will turn out to be a large number and therefore the value of *y* will be close to 0. Graphically, the sigmoid function presents this form:

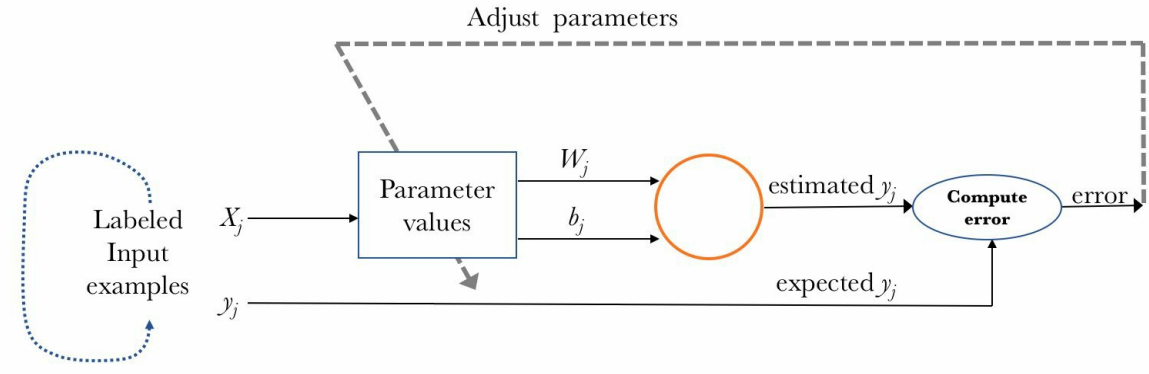


So far we have presented how to define an artificial neuron, the simplest architecture that a neural network can have. In particular this architecture is named in the literature of the subject as Perceptron[115] (also called *linear threshold unit* (LTU)), invented in 1957 by Frank Rosenblatt, and visually summarized in a general way with the following scheme:



Finally, let me help the reader to intuit how this neuron can learn the weights *W* and the biases *b* from the input data that we already have labeled as "squares" or "circles" (in chapter 4 we will present how this process is done in a more formal way).

It is an iterative process for all the known labeled input examples, comparing the value of its label estimated through the model, with the expected value of the label of each element. After each iteration, the parameter values are adjusted in such a way that the error obtained is getting smaller as we keep on iterating with the goal of minimizing the loss function mentioned above. The following scheme wants to visually summarize the learning process of one perceptron in a general way:

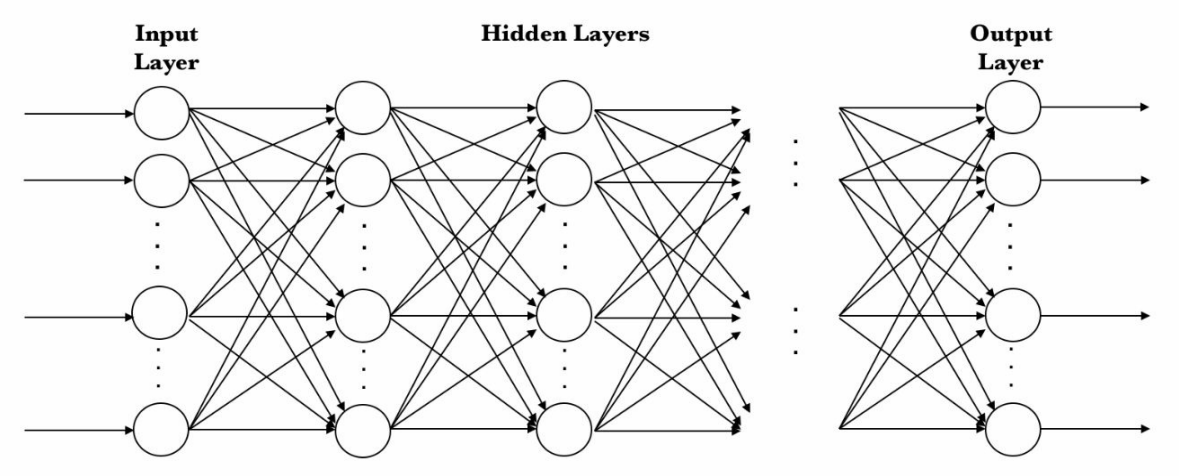


Multi-Layer Perceptron

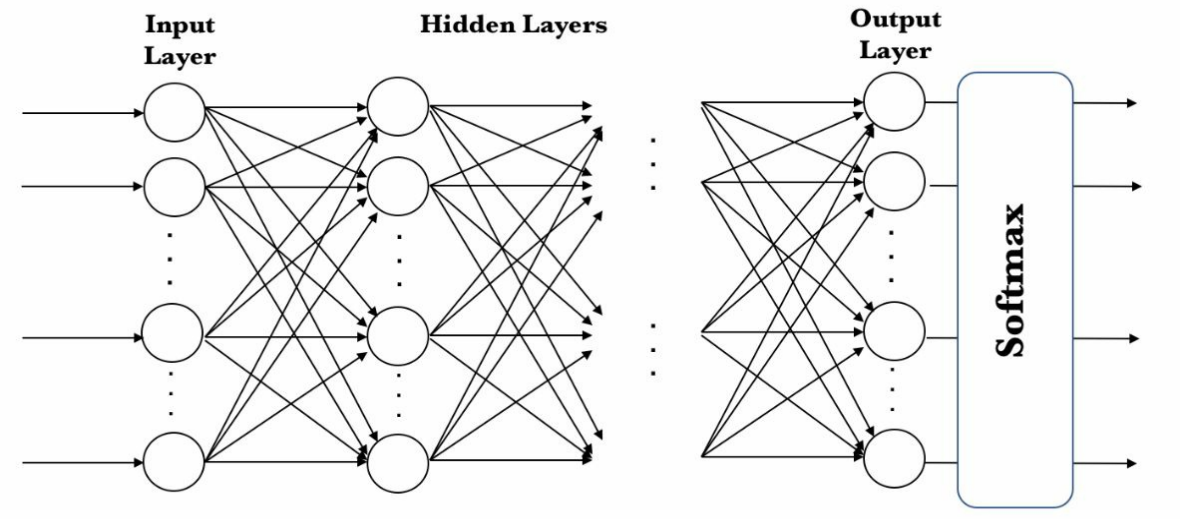
But before moving forward with the example, we will briefly introduce the form that neural networks usually take when they are constructed from perceptrons like the one we have just presented.

In the literature of the area we refer to a Multi-Layer Perceptron (MLP) when we find neural networks that have an *input layer*, one or more layers composed of perceptrons, called *hidden layers* and a final layer with several

perceptrons called the *output layer*. In general we refer to *Deep Learning* when the model based on neural networks is composed of multiple hidden layers. Visually it can be presented with the following scheme:



MLPs are often used for classification, and specifically when classes are exclusive, as in the case of the classification of digit images (in classes from 0 to 9). In this case, the output layer returns the probability of belonging to each one of the classes, thanks to a function called softmax. Visually we could represent it in the following way:



As we will present in chapter 4, there are several activation functions in

addition to the sigmoid, each with different properties. One of them is the one we just mentioned, the *softmax* activation function[116], which will be useful to present an example of a simple neural network to classify in more than two classes. For the moment we can consider the softmax function as a generalization of the sigmoid function that allows us to classify more than two classes.

3 Softmax activation function

We will solve the problem in a way that, given an input image, we will obtain the probabilities that it is each of the 10 possible digits. In this way, we will have a model that, for example, could predict a nine in an image, but only being sure in 80% that it is a nine. Due to the stroke of the bottom of the number in this image, it seems that it could become an eight in a 5% chance and it could even give a certain probability to any other number. Although in this particular case we will consider that the prediction of our model is a nine since it is the one with the highest probability, this approach of using a probability distribution can give us a better idea of how confident we are of our prediction. This is good in this case, where the numbers are made by hand, and surely in many of them, we cannot recognize the digits with 100% certainty.

Therefore, for this example of MNIST classification we will obtain, for each input example, an output vector with the probability distribution over a set of mutually exclusive labels. That is, a vector of 10 probabilities each corresponding to a digit and also the sum of all these 10 probabilities results in the value of 1 (the probabilities will be expressed between 0 and 1).

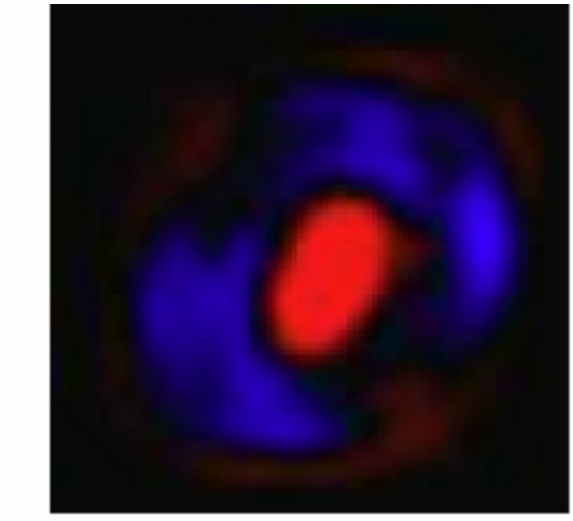
As we have already advanced, this is achieved through the use of an output layer in our neural network with the softmax activation function, in which each neuron in this softmax layer depends on the outputs of all the other neurons in the layer, since that the sum of the output of all of them must be 1.

But how does the softmax activation function work? The softmax function is based on calculating "the evidence" that a certain image belongs to a particular class and then these evidences are converted into probabilities that it belongs to each of the possible classes.

An approach to measure the evidence that a certain image belongs to a particular class is to make a weighted sum of the evidence of belonging to

each of its pixels to that class. To explain the idea I will use a visual example.

Let's suppose that we already have the model learned for the number zero (we will see later how these models are learned). For the moment, we can consider a model as "something" that contains information to know if a number is of a certain class. In this case, for the number zero, suppose we have a model like the one presented below:

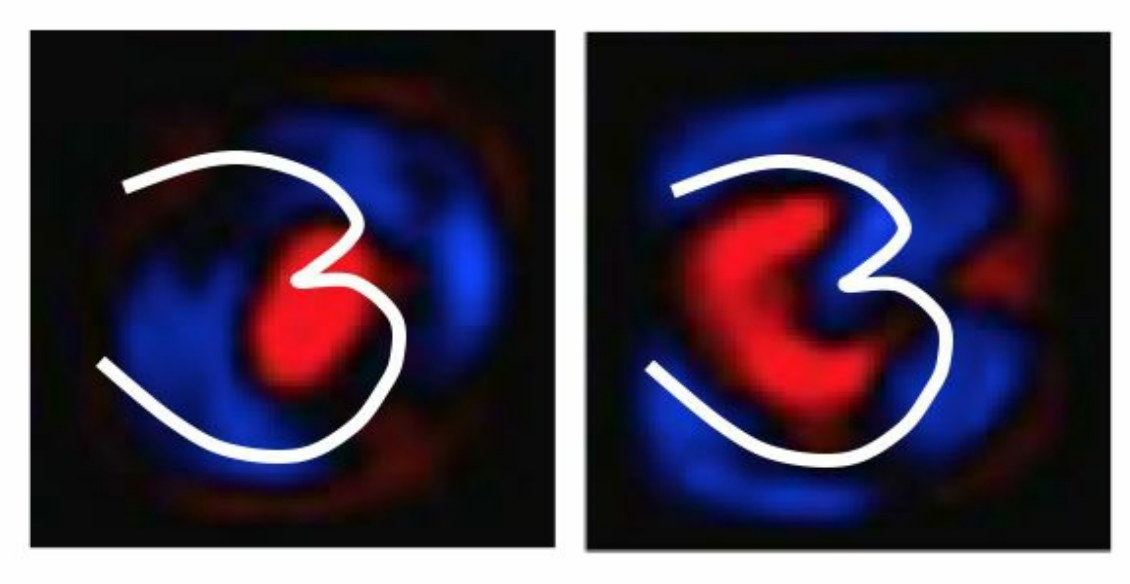


In this case, with a matrix of 28×28 pixels, where the pixels in red (in the white/black edition of the book is the lightest gray) represent negative weights (i.e., reduce the evidence that it belongs), while that the pixels in blue (in the black/white edition of the book is the darkest gray) represent positive weights (the evidence of which is greater increases). The black color represents the neutral value.

Let's imagine that we trace a zero over it. In general, the trace of our zero would fall on the blue zone (remember that we are talking about images that have been normalized to 20×20 pixels and later centered on a 28×28 image). It is quite evident that if our stroke goes over the red zone, it is most likely that we are not writing a zero; therefore, using a metric based on adding if we pass through the blue zone and subtracting if we pass through the red zone

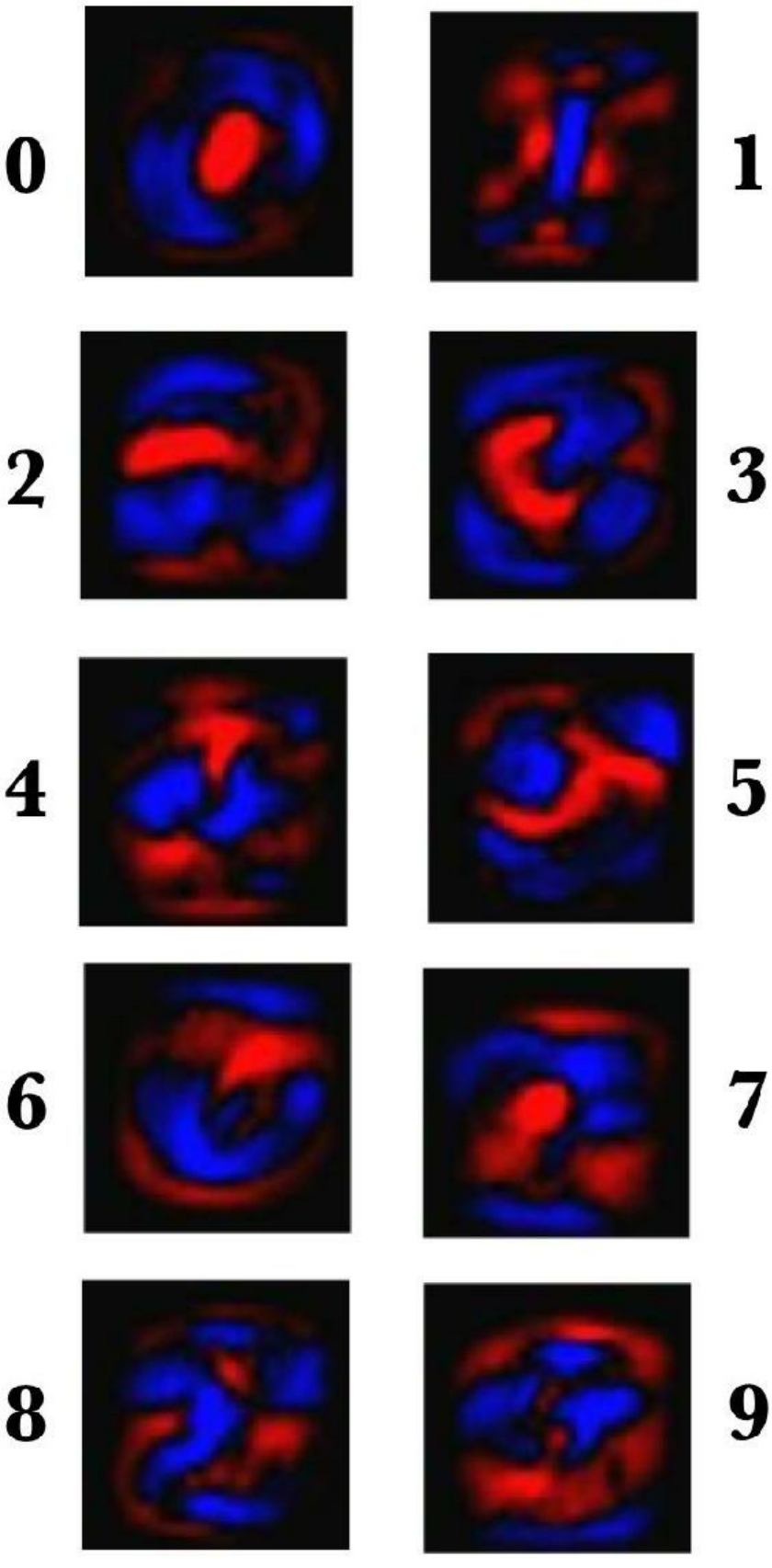
seems reasonable.

To confirm that it is a good metric, let's imagine now that we draw a three; it is clear that the red zone of the center of the previous model that we used for the zero will penalize the aforementioned metric since, as we can see in the left part of this figure, when writing a three we pass over:



But on the other hand, if the reference model is the one corresponding to number 3 as shown in the right part of the previous figure, we can see that, in general, the different possible traces that represent a three are mostly maintained in the blue zone.

I hope that the reader, seeing this visual example, already intuits how the approximation of the weights indicated above allows us to estimate what number it is.


0
1
2
3
4
5
6
7
8
9

The previous figure shows the weights of a concrete model example learned for each of these ten MNIST classes (figure obtained from the example of the Tensorflow tutorial[117]). Remember that we have chosen red (lighter gray in black and white book edition) in this visual representation for negative weights, while we will use blue (darker gray in black and white book edition) to represent positives[118].

Once the evidence of belonging to each of the 10 classes has been calculated, these must be converted into probabilities whose sum of all their components add 1. For this, softmax uses the exponential value of the calculated evidence and then normalizes them so that the sum equates to one, forming a probability distribution. The probability of belonging to class *i* is:

$$Softmax_i = \frac{e^{evidence_i}}{\sum_j e^{evidence_j}}$$

Intuitively, the effect obtained with the use of exponentials is that one more unit of evidence has a multiplier effect and one unit less has the inverse effect. The interesting thing about this function is that a good prediction will have a single entry in the vector with a value close to 1, while the remaining entries will be close to 0. In a weak prediction, there will be several possible labels, which will have more or less the same probability.

Case Study with Keras

This chapter will show how the example of MNIST is encoded with Keras to offer the reader a first contact with this library and understand the structure that the implementation of this example has with Keras. But first let's take the opportunity to explain some interesting details about the available data.

Data to feed a neural network

Dataset for training, validation and testing

Before presenting the implementation in Keras of the previous example, let's review how we should distribute the available data in order to configure and evaluate the model correctly.

For the configuration and evaluation of a model in Machine Learning, and therefore Deep Learning, the available data are usually divided into three sets: training data, validation data, and test data. The training data are those used for the learning algorithm to obtain the parameters of the model with the iterative method that we have already mentioned.

If the model does not completely adapt to the input data (for example, if it presented overfitting), in this case, we would modify the value of certain hyperparameters and after training it again with the training data we would evaluate it again with the validation ones. We can make these adjustments of the hyperparameters guided by the validation data until we obtain validation results that we consider correct. If we have followed this procedure, we must be aware that, in fact, the validation data have influenced the model so that it also fits the validation data. For this reason, we always reserve a set of test data for final evaluation of the model that will only be used at the end of the whole process, when we consider that the model is already fine-tuned and we will no longer modify any of its hyperparameters.

Given the introductory nature of this book and that we will not go into detail of tuning the hyperparameters, in the examples we will ignore the validation data and only use the training and test data.

Preloaded data in Keras

In Keras the MNIST dataset is preloaded in the form of four Numpy arrays

and can be obtained with the following code:

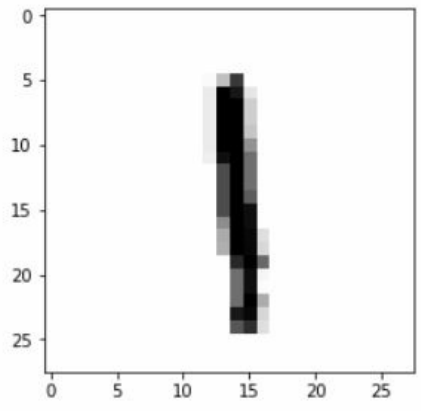
```
import keras

from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

x_train and *y_train* contain the training set, while *x_test* and *y_test* contain the test data. The images are encoded as Numpy arrays and their corresponding labels ranging from 0 to 9. Following the strategy of the book to gradually introduce the concepts of the subject, as we have indicated, we will not see yet how to separate a part of the training data to use them as Validation data. We will only take into account the training and test data.

If we want to check what values we have loaded, we can choose any of the images of the MNIST set, for example image 8, and using the following Python code:

```
import matplotlib.pyplot as plt
plt.imshow(x_train[8], cmap=plt.cm.binary)
```

We get the following image:



And if we want to see its corresponding label we can do it through:

```
print(y_train[8])
```

```
1
```

That, as we see, it returns the value of "1", as expected.

Data representation in Keras

Keras, which as we have seen uses a multidimensional array of Numpy as a basic data structure, calls this data structure a *tensor*. In short, we could say that a tensor has three main attributes:

- *Number of axes* (*Rank*): a tensor containing a single number will be called scalar (or a 0-dimensional tensor, or tensor 0D). An array of numbers we call vector, or tensor 1D. An array of vectors will be a matrix, or 2D tensor. If we pack this matrix in a new array, we get a 3D tensor, which we can interpret visually as a cube of numbers. By packaging a 3D tensioner in an array, we can create a 4D tensioner, and so on. In the Python Numpy library this is called the *tensor's ndim*.
- *Shape*: it is a tuple of integers that describe how many dimensions

the tensor has along each axis. In the Numpy library this attribute is called *shape*.

- D*ata type*: this attribute indicates the type of data that contains the tensor, which can be for example *uint8*, *float32*, *float64*, etc. In the Numpy library this attribute is called *dtype*.

I propose that we obtain the number of axes and dimensions of the tensor *train_images* from our previous example:

```
print(x_train.ndim)
```

```
3
```

```
print(x_train.shape)
```

```
(60000, 28, 28)
```

And if we want to know what type of data it contains:

```
print(x_train.dtype)
```

```
uint8
```

Data normalization in Keras

These MNIST images of 28×28 pixels are represented as an array of numbers whose values range from [0, 255] of type uint8. But it is usual to scale the input values of neural networks to certain ranges. In the example of this chapter the input values should be scaled to values of type float32 within the interval [0, 1]. We can achieve this transformation with the following lines of code:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

On the other hand, to facilitate the entry of data into our neural network (we will see that in convolutionals it is not necessary) we must make a transformation of the tensor (image) from 2 dimensions (2D) to a vector of 1 dimension (1D). That is, the matrix of 28×28 numbers can be represented by a vector (array) of 784 numbers (concatenating row by row), which is the format that accepts as input a densely connected neural network like the one we will see in this chapter. In Python, converting every image of the MNIST dataset to avector with 784 components can be accomplished as follows:

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

After executing these Python instructions, we can verify that *x_train.shape* takes the form of (60000, 784) and *x_test.shape* takes the form of (10000, 784), where the first dimension indexes the image and the second indexes the pixel in each image (now the intensity of the pixel is a value between 0 and 1):

```
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

In addition to that, we have the labels for each input data (remember that in our case they are numbers between 0 and 9 that indicate which digit represents the image, that is, to which class is associated). In this example, and as we have already advanced, we will represent this label with a vector of 10 positions, where the position corresponding to the digit that represents the image contains a 1 and the remaining positions of the vector contain the value 0.

In this example we will use what is known as *one-hot encoding*, which we have already mentioned, which consists of transforming the labels into a vector of as many zeros as the number of different labels, and containing the value of 1 in the index that corresponds to the value of the label. Keras offers many support functions, including *to_categorical* to perform precisely this transformation, which we can import from *keras.utils*:

```
from keras.utils import to_categorical
```

To see the effect of the transformation we can see the values before and after applying *to_categorical* :

```
print(y_test[0])
```

```
7
```

```
print(y_train[0])
```

```
5
```

```
print(y_train.shape)
```

```
(60000,)
```

```
print(x_test.shape)
```

```
(10000, 784)
```

```
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

print(y_test[0])
```

```
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

```
print(y_train[0])
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

```
print(y_train.shape)
```

```
(60000, 10)
```

```
print(y_test.shape)
```

```
(10000, 10)
```

Now we have the data ready to be used in our simple model example that we are going to program in Keras in the next section.

2 Densely connected networks in Keras

In this section, we will present how to specify in Keras the model that we have defined in the previous sections.

Sequential class in Keras

The main data structure in Keras is the *Sequential* class, which allows the creation of a basic neural network. Keras also offers an API[119] that allows implementing more complex models in the form of a graph that can have multiple inputs, multiple outputs, with arbitrary connections in between, but it is beyond the scope of this book.

The *Sequential*[120] class of the Keras library is a wrapper for the sequential neural network model that Keras offers and can be created in the following way:

```
from keras.models import Sequential
model = Sequential()
```

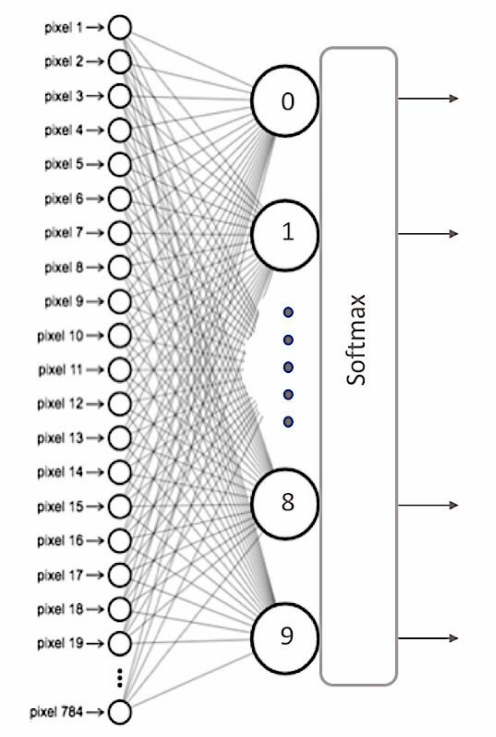
In this case, the model in Keras is considered as a sequence of layers and each of them gradually "distills" the input data to obtain the desired output. In Keras we can find all the required types of layers that can be easily added to the model through the *add()* method.

Defining the model

The construction in Keras of our model to recognize the images of digits could be the following:

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation

model = Sequential()
model.add(Dense(10, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

Here, the neural network has been defined as a sequence of two layers that are densely connected (or fully connected), meaning that all the neurons in each layer are connected to all the neurons in the next layer. Visually we could represent it in the following way:



In the previous code we explicitly express in the *input_shape* argument of the first layer what the input data is like: a tensor that indicates that we have 784 features of the model (in fact the tensor that is being defined is *(None, 784,)* as we will see more ahead).

A very interesting characteristic of the Keras library is that it will automatically deduce the shape of the tensors between layers after the first one. This means that the programmer only has to establish this information for the first of them. Also, for each layer we indicate the number of nodes that it has and the activation function that we will apply in it (in this example, *sigmoid*).

The second layer in this example is a *softmax* layer of 10 neurons, which means that it will return a matrix of 10 probability values representing the 10 possible digits (in general, the output layer of a classification network will have as many neurons as classes, except in a binary classification, where only one neuron is needed). Each value will be the probability that the image of the current digit belongs to each one of them.

A very useful method that Keras provides to check the architecture of our model is *summary()*:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                7850
_________________________________________________________________
dense_2 (Dense)              (None, 10)                110
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
```

Later we will go into more detail with the information that returns the *summary()* method, because this calculation of parameters and sizes of the data that the neural network has when we start to build very large network

models is very valuable. For our simple example, we see that it indicates that 7,960 parameters are required (column *Param #*), which correspond to 7,850 parameters to the first layer and 110 to the second.

In the first layer, for each neuron i (between 0 and 9) we require 784 parameters for the weights w_{ij} and therefore 10×784 parameters to store the weights of the 10 neurons. In addition to the 10 additional parameters for the 10 b_j biases corresponding to each one of them. In the second layer, being a softmax function, it is required to connect all 10 neurons with the 10 neurons of the previous layer. Therefore 10x10 w_i parameters are required and in addition 10 b_j biases corresponding to each node.

The details of the arguments that we can indicate for the *Dense*[121] layer can be found in the Keras manual. In our example, the most relevant ones appear. The first argument indicates the number of neurons in the layer; the following is the activation function that we will use in it. In the next chapter we will discuss in more detail other possible activation functions beyond the two presented here: sigmoid and softmax.

The initialization of the weights is also often indicated as an argument of the *Dense* layers. The initial values must be adequate for the optimization problem to converge as quickly as possible. The various initialization options[122] can also be found in the Keras manual.

3 Basic steps to implement a neural network in Keras

Next, we will present a brief description of the steps we must perform to implement a basic neural network and, in the following chapters, we will gradually introduce more details about each of these steps.

Configuration of the learning process

From the *Sequential* model, we can define the layers in a simple way with the *add()* method, as we have advanced in the previous section. Once we have our model defined, we can configure how its learning process will be with the *compile()* method, with which we can specify some properties through method arguments.

The first of these arguments is the *loss function* that we will use to evaluate the degree of error between calculated outputs and the desired outputs of the training data. On the other hand, we specify an *optimizer* that, as we will see, is the way we have to specify the optimization algorithm that allows the neural network to calculate the weights of the parameters from the input data and the defined loss function. More detail of the exact purpose of the loss function and the optimizer will be presented in the next chapter.

And finally we must indicate the metric that we will use to monitor the learning process (and test) of our neural network. In this first example we will only consider the *accuracy* (fraction of images that are correctly classified). For example, in our case we can specify the following arguments in *compile()* method to test it on our computer:

```
model.compile(loss="categorical_crossentropy",
              optimizer="sgd",
              metrics = ['accuracy'])
```

In this example we specify that the loss function is *categorical_crossentropy*, the optimizer used is the *stocastic gradient descent (sgd)* and the metric is *accuracy*, with which we will evaluate the percentage of correct guesses.

Model training

Once our model has been defined and the learning method configured, it is ready to be trained. For this we can train or "adjust" the model to the training data available by invoking the *fit()* method of the model:

```
model.fit(x_train, y_train, batch_size=100, epochs=5)
```

In the first two arguments we have indicated the data with which we will train the model in the form of Numpy arrays. The *batch_size* argument indicates the number of data that we will use for each update of the model parameters and with *epochs* we are indicating the number of times we will use all the data in the learning process. These last two arguments will be explained in much more detail in the next chapter.

This method finds the value of the parameters of the network through the iterative training algorithm that we mentioned and we will present in a bit more detail in the next chapter. Roughly, in each iteration of this algorithm,

this algorith takes training data from *x_train*, passes them through the neural network (with the values that their parameters have at that moment), compares the obtained result with the expected one (indicated in *y_train*) and calculates the *loss* to guide the adjustment process of the model parameters, which intuitively consists of applying the optimizer specified above in the *compile()* method to calculate a new value of each one of the model parameters (weights and biases)in each iteration in such a way that the loss is reduced.

This is the method that, as we will see, may take longer and Keras allows us to see its progress using the *verbose* argument (by default, equal to 1), in addition to indicating an estimate of how long each *epoch* takes:

```
Epoch 1/5
60000/60000 [==============================] - 1s 15us/step - loss: 2.1822 - acc: 0.2916
Epoch 2/5
60000/60000 [==============================] - 1s 12us/step - loss: 1.9180 - acc: 0.5283
Epoch 3/5
60000/60000 [==============================] - 1s 13us/step - loss: 1.6978 - acc: 0.5937
Epoch 4/5
60000/60000 [==============================] - 1s 14us/step - loss: 1.5102 - acc: 0.6537
Epoch 5/5
60000/60000 [==============================] - 1s 13us/step - loss: 1.3526 - acc: 0.7034
10000/10000 [==============================] - 0s 22us/step
```

This is a simple example so that the reader at the end of the chapter has already been able to program their first neural network but, as we will see, the *fit()* method allows many more arguments that have a very important impact on the learning outcome. Furthermore, this method returns a *History* object that we have omitted in this example. Its *History.history* attribute is the record of the *loss* values for the training data and other metrics in successive *epochs*, as well as other metrics for the validation data if they have been specified.

Model evaluation

At this point, the neural network has been trained and its behavior with new test data can now be evaluated using the *evaluation()* method. This method returns two values:

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

These values indicate how well or badly our model behaves with new data that it has never seen. These data have been stored in *x_test* and *y_test* when we have performed the *mnist.load_data()* and we pass them to the method as arguments. In the scope of this book we will only look at one of them, the accuracy:

```
print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.9018
```

The accuracy is telling us that the model we have created in this chapter, applied to data that the model has never seen before, classifies 90% of them correctly.

The reader should note that, in this example, to evaluate the model we have only focused on its accuracy, that is, the ratio between the correct predictions that the model has made over the total predictions regardless of what category it is. However, although in this case it is sufficient, sometimes it is necessary to delve a little more and take into account the types of correct and incorrect

predictions made by the model in each of its categories.

In Machine Learning, a very useful tool for evaluating models is the confusion matrix, a table with rows and columns that count the predictions in comparison with the real values. We use this table to better understand how well the model behaves and it is very useful to show explicitly when one class is confused with another. A confusion matrix for a binary classifier like the one explained in the previous chapter has this structure:

		Predicted class	
		positive	negative
Actual class	positive	TP	FN
	negative	FP	TN

True positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), are the four different possible outcomes of a single prediction for a two-class case with classes “1” (“positive”) and “0” (“negative”).

A false positive is when the outcome is incorrectly classified as positive,

when it is in fact negative. A false negative is when the outcome is incorrectly classified as negative when it is in fact positive. True positives and true negatives are obviously correct classifications.

With this confusion matrix, the accuracy can be calculated by adding the values of the diagonal and dividing them by the total:

$$Accuracy = (TP + TN) / (TP + FP + FN + TN)$$

Nonetheless, the accuracy can be misleading in terms of the quality of the model because, when measuring it for the concrete model, we do not distinguish between the false positive and false negative type errors, as if both had the same importance. For example, think of a model that predicts if a mushroom is poisonous. In this case, the cost of a false negative, that is, a poisonous mushroom given for consumption could be dramatic. On the contrary, a false positive has a very different cost.

For this reason we have another metric called *Sensitivity* (or *recall*) that tells us how well the model avoids false negatives:

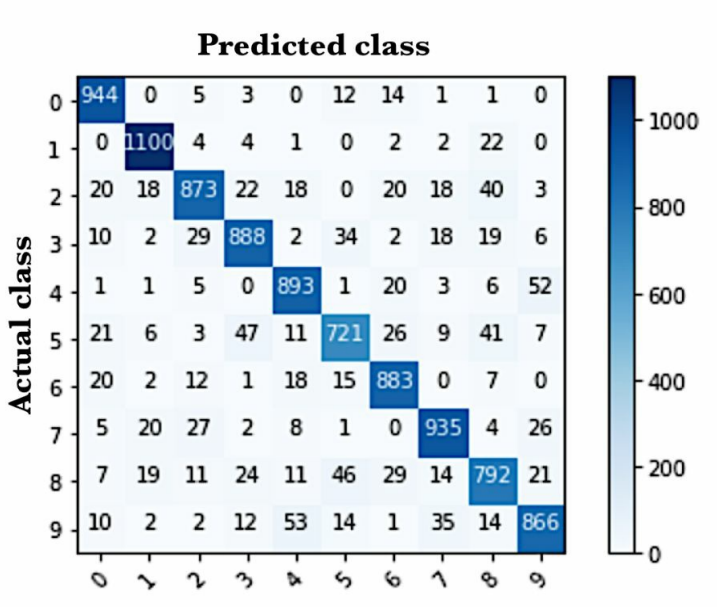
$$Sensitivity = TP / (TP + FN)$$

In other words, from the total of positive observations (poisonous mushrooms), how many the model detects.

From the confusion matrix, several metrics can be obtained to focus other cases as shown in this link[123], but it is beyond the scope of this book to enter more in detail on this topic. The convenience of using one metric or another will depend on each particular case and, in particular, the "cost" associated

with each classification error of the model.

But the reader will wonder how is this confusion matrix in our classifier, where there are 10 classes instead of 2. In this case, I suggest using the *Scikit-learn*[124] package to evaluate the quality of the model by calculating the confusion matrix, presented in the following figure[125]:



In this case, the elements of the diagonal represent the number of points in which the label predicted by the model coincides with the actual value of the label, while the other values indicate the cases in which the model has classified incorrectly. Therefore, the higher the values of the diagonal, the better the prediction will be. In this example, if the reader calculates the sum of the values of the diagonal divided by the total values of the matrix, he or she will see that it matches the accuracy that the *evaluate()* method has returned.

In the GitHub of the book, the reader can find the code used to calculate this confusion matrix.

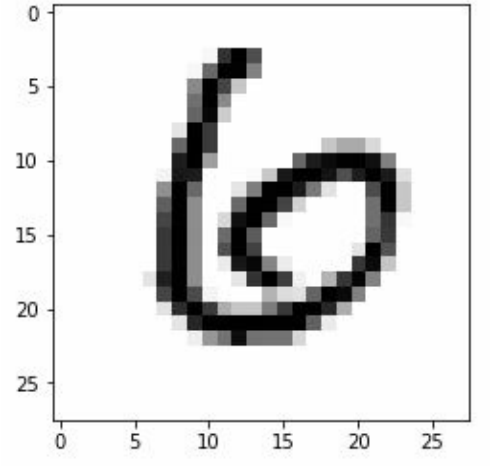
Generate predictions

Finally, readers need to know how we can use the model trained in the previous section to make predictions. In our example, it consists in predict which digit represents an image. In order to do this, Keras supply the *predict()* method.

To test this method we can choose any element. For ease, let's take one from the test dataset *x_test*. For example let's choose the element 11 of this dataset *x_test*.

Before seeing the prediction, let's see the image to be able to check ourselves if the model is making a correct prediction (before doing the previous reshape):

```
plt.imshow(x_test[11], cmap=plt.cm.binary)
```



I think the reader will agree that in this case it corresponds to number 6.

Now let's see that the *predict()* method of the model, executing the following code, correctly predicts the value that we have just estimated that it should predict.

```
predictions = model.predict(x_test)
```

The predict() method return a vector with the predictions for the whole dataset elements. We can know which class gives the most probability of belonging by means of the argmax function of Numpy, which returns the index of the position that contains the highest value of the vector. Specifically, for item 11:

```
np.argmax(predictions[11])
```

```
6
```

We can check it printing the vector returned by the method:

```
print(predictions[11])
```

```
[0.06 0.01 0.17 0.01 0.05 0.04 0.54 0.   0.11 0.02]
```

We see that the highest value in the vector is in the position 6. We can also verify that the result of the prediction is a vector whose sum of all its components is equal to 1, as expected. For this we can use:

```
np.sum(predictions[11])
```

1.0

So far the reader has been able to create their first model in Keras that correctly classifies the MNIST digits 90% of the time. In the next chapter, we will present how the learning process works and several of the hyperparameters that we can use in a neural network to improve these results. In chapter 6 we will see how we can improve these classification results using convolutional neural networks for the same example.

Some basics about the learning process

In this chapter we will present an intuitive vision of the main components of the learning process of a neural network. We will also see some of the most relevant parameters and hyperparameters in Deep Learning.

Learning process of a neural network

Remember that a neural network is made up of neurons connected to each other; at the same time, each connection of our neural network is associated with a weight that dictates the importance of this relationship in the neuron when multiplied by the input value.

Each neuron has an **activation function** that defines the output of the neuron. The activation function is used to introduce non-linearity in the modeling capabilities of the network. We have several options for activation functions that we will present in this chapter.

Training our neural network, that is, learning the values of our parameters (weights w_{ij} and b_j biases) is the most genuine part of Deep Learning and we can see this learning process in a neural network as an iterative process of "going and return" by the layers of neurons. The "going" is a forwardpropagation of the information and the "return" is a backpropagation of the information.

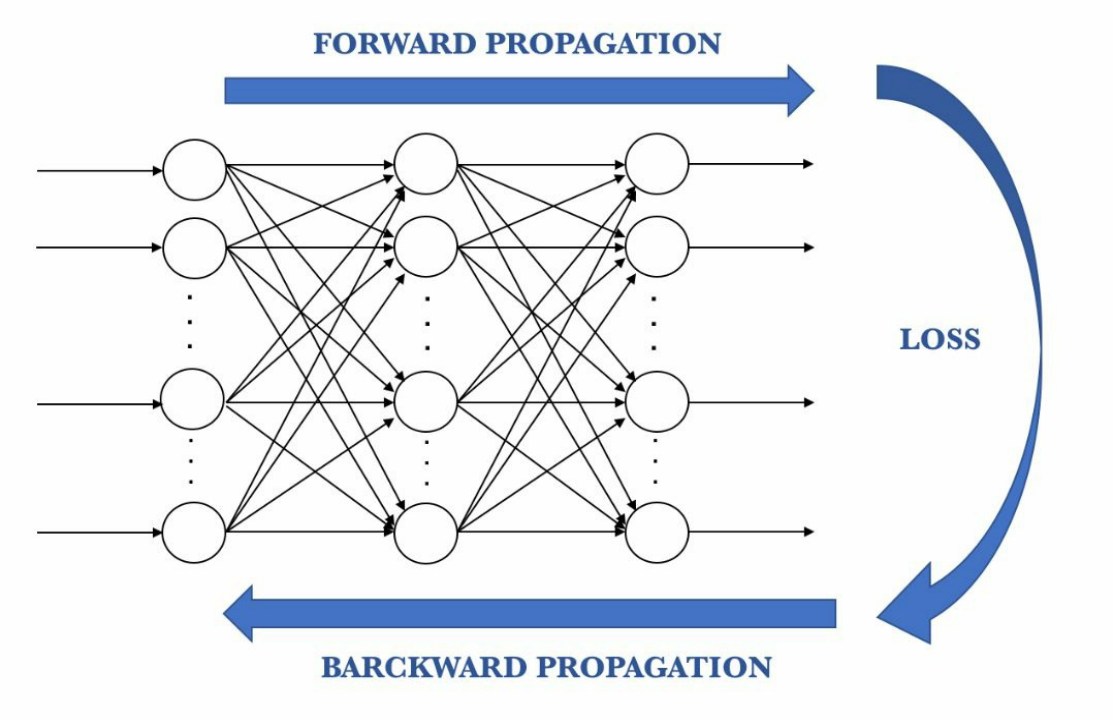
The first phase **forwardpropagation** occurs when the network is exposed to the training data and these cross the entire neural network for their predictions (labels) to be calculated. That is, passing the input data through the network in such a way that all the neurons apply their transformation to the information they receive from the neurons of the previous layer and sending it to the neurons of the next layer. When the data has crossed all the layers, and all its neurons have made their calculations, the final layer will be reached with a result of label prediction for those input examples.

Next, we will use a **loss function** to estimate the loss (or error) and to compare and measure how good/bad our prediction result was in relation to the correct result (remember that we are in a supervised learning environment and we have the label that tells us the expected value). Ideally, we want our cost to be zero, that is, without divergence between estimated and expected

value. Therefore, as the model is being trained, the weights of the interconnections of the neurons will gradually be adjusted until good predictions are obtained.

Once the loss has been calculated, this information is propagated backwards. Hence, its name: **backpropagation**. Starting from the output layer, that loss information propagates to all the neurons in the hidden layer that contribute directly to the output. However, the neurons of the hidden layer only receive a fraction of the total signal of the loss, based on the relative contribution that each neuron has contributed to the original output. This process is repeated, layer by layer, until all the neurons in the network have received a loss signal that describes their relative contribution to the total loss.

Visually, we can summarize what we have explained with this visual scheme of the stages:



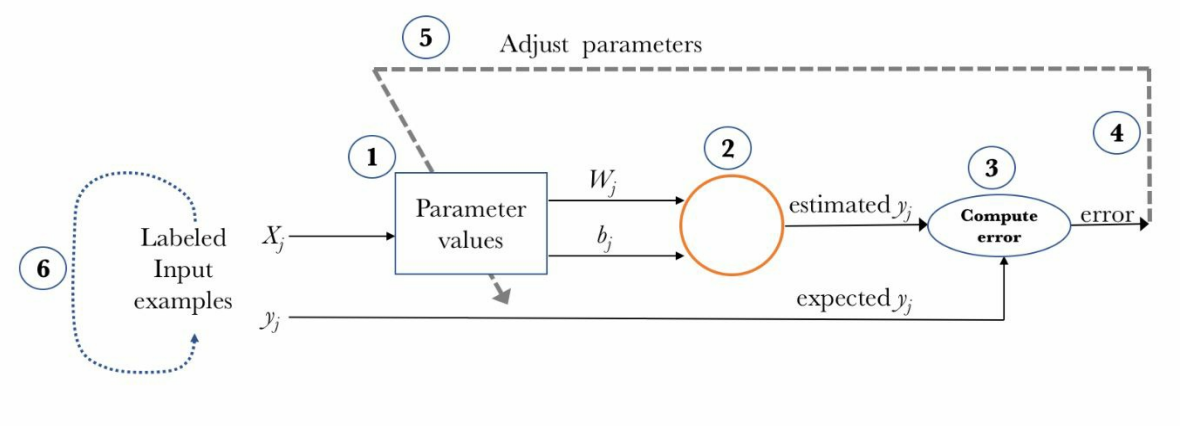
Now that we have spread this information back, we can adjust the weights of connections between neurons. What we are doing is making the loss as close as possible to zero the next time we go back to using the network for a

prediction. For this, we will use a technique called **gradient descent**. This technique changes the weights in small increments with the help of the calculation of the derivative (or gradient) of the loss function, which allows us to see in which direction "to descend" towards the global minimum; this is done in general in batches of data in the successive iterations (epochs) of all the dataset that we pass to the network in each iteration.

To recap, the learning algorithm consists of:

1. Start with values (often random) for the network parameters (w_{ij} weights and b_j biases)
2. Take a set of examples of input data and pass them through the network to obtain their prediction.
3. Compare these predictions obtained with the values of expected labels and calculate the loss with them.
4. Perform the backpropagation in order to propagate this loss to each and every one of the parameters that make up the model of the neural network.
5. Use this propagated information to update the parameters of the neural network with the gradient descent in a way that the total loss is reduced and a better model is obtained.
6. Continue iterating in the previous steps until we consider that we have a good model.

Returning to the figure in chapter 2 which visually summarizes the learning process of one perceptron in a general way, these stages would be reflected in this way:



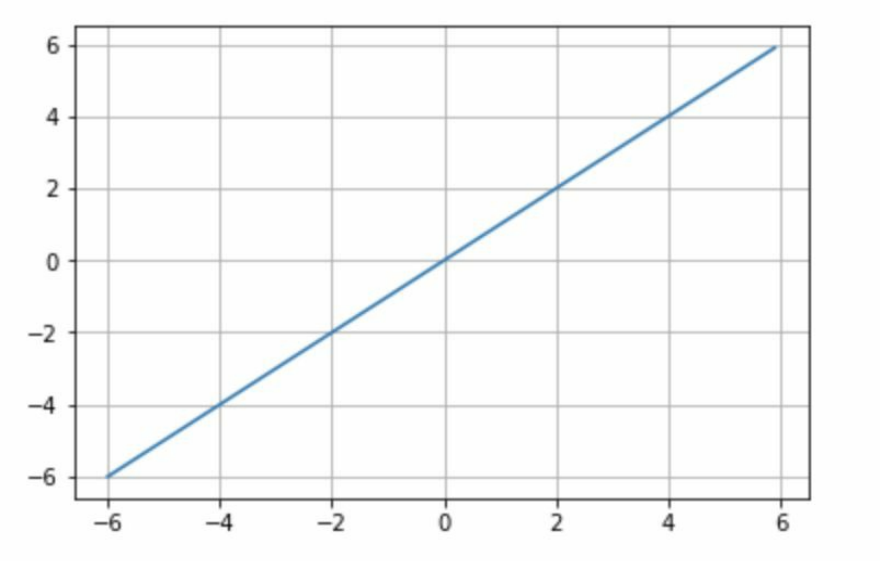
Below we present in more detail each of the elements that we have highlighted in this section.

2 Activation functions

Remember that we use the activation functions to propagate the output of a neuron forward. This output is received by the neurons of the next layer to which this neuron is connected (up to the output layer included). As we have said, the activation function serves to introduce non-linearity in the modeling capabilities of the network. Below we will list the most used nowadays; all of them can be used in a layer of Keras (we can find more information on their website[126]).

Linear

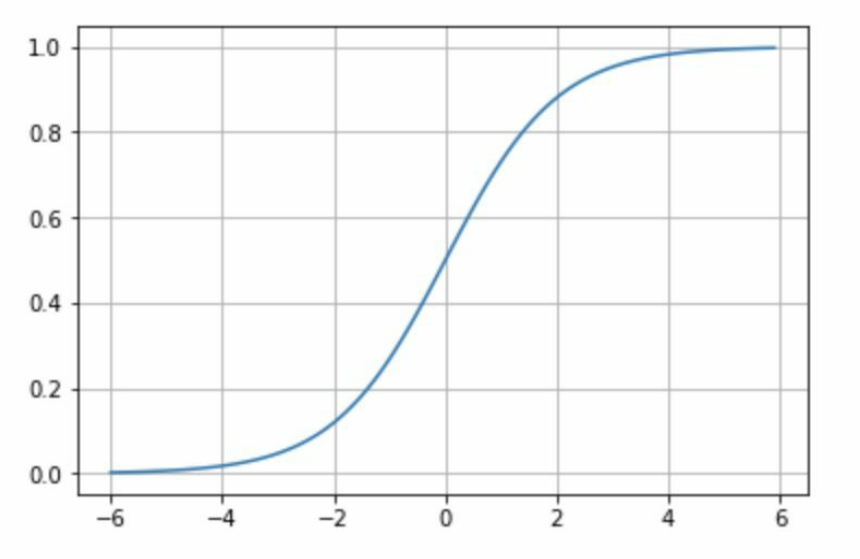
The linear activation function is basically the identity function in which, in practical terms, it means that the signal does not change.



Sigmoid

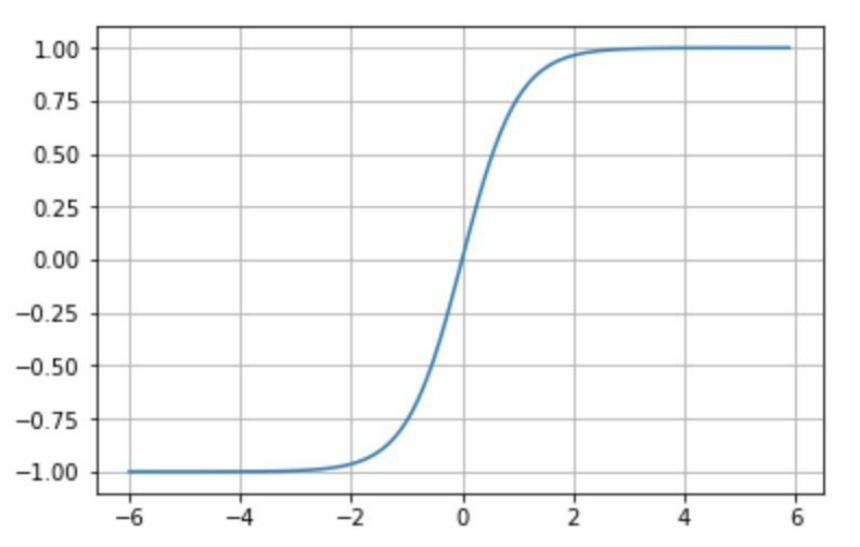
The sigmoid function has already been introduced in chapter 2. Its interest lies in the fact that it allows a reduction in extreme or atypical values in valid

data without eliminating them: it converts independent variables of almost infinite range into simple probabilities between 0 and 1. Most of its output will be very close to the extremes of 0 or 1.



Tanh

Without going into detail, we can summarize that the tanh represents the relationship between the hyperbolic sine and the hyperbolic cosine: tanh (x) = sinh (x)/cosh (x) [127]. Unlike the sigmoid function, the normalized range of tanh is between -1 and 1, which is the input that goes well with some neural networks. The advantage of tanh is that negative numbers can be dealt with more easily.

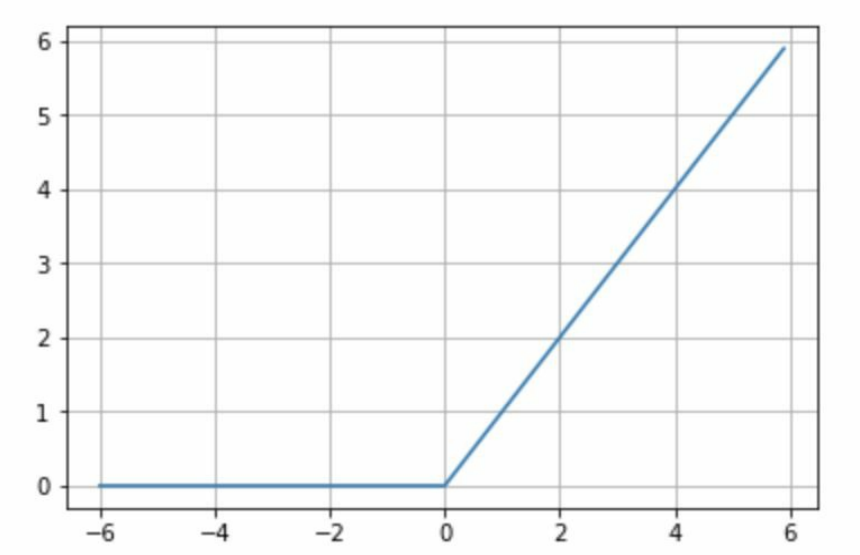


Softmax

The softmax activation function was also presented in chapter 2 to generalize the logistic regression, insofar as instead of classifying in binary it can contain multiple decision limits. As we have seen, the softmax activation function will often be found in the output layer of a neural network and return the probability distribution over mutually exclusive output classes.

ReLU

The activation function rectified linear unit (ReLU) is a very interesting transformation that activates a single node if the input is above a certain threshold. The default and more usual behavior is that, as long as the input has a value below zero, the output will be zero but, when the input rises above, the output is a linear relationship with the input variable of the form $f(x) = x$. The ReLU activation function has proven to work in many different situations and is currently widely used.


6
5
4
3
2
1
0
−6
−4
−2
0
2
4
6

In summary, we can consider backpropagation as a method to alter the parameters (weights and biases) of the neural network in the right direction. It starts by calculating the loss term first, and then the parameters of the neural network are adjusted in reverse order with an optimization algorithm taking into account this calculated loss.

Remember that in Keras the *compile()* method allows us to define how we want the components that are involved in the learning process to be:

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Specifically, in this example, three arguments are passed to the method: an optimizer, a loss function, and a list of metrics. In classification problems like our example, accuracy is used as a metric. Let's go a little deeper into these arguments.

Loss function

A loss function is one of the parameters required to quantify how close a particular neural network is to the ideal weight during the training process.

In the Keras manual page[128], we can find all types of loss functions available. Some have their concrete hyperparameters that must be indicated; in the example of the previous chapter, when we use *categorical_crossentropy* as a function of loss, our output must be in a categorical format. The choice of the best function of loss resides in understanding what type of error is or is not acceptable for the problem in particular.

Optimizers

The optimizer is another of the arguments required in the *compile()* method. Keras currently has different optimizers that can be used: *SGD*, *RMSprop*, *Adagrad*, *Adadelta*, *Adam*, *Adamax*, *Nadam*. You can find more detail about each of them in the Keras documentation[129].

In general, we can see the learning process as a global optimization problem where the parameters (weights and biases) must be adjusted in such a way that the loss function presented above is minimized. In most cases, these parameters cannot be solved analytically, but in general they can be approached well with iterative or optimizing optimization algorithms, such as those mentioned above.

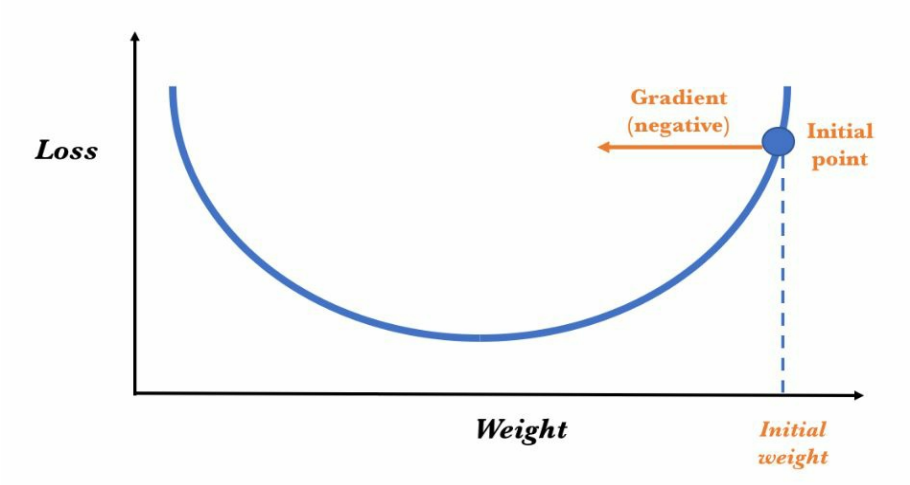
Gradient descent

We will explain one of the concrete optimizers so that you understand the overall operation of the optimizers. Specifically, the *gradient descent*, the basis of many optimizers and one of the most common optimization algorithms in Machine Learning and Deep Learning.

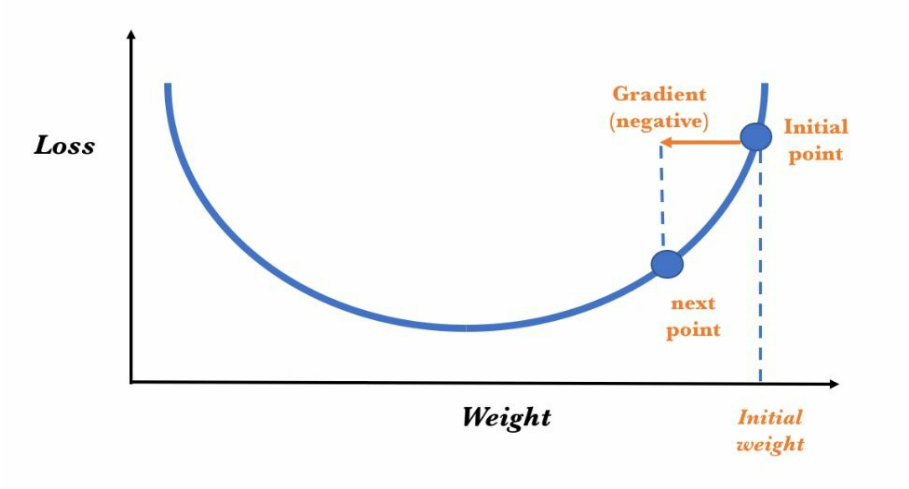
Gradient descent uses the first derivative (gradient) of the loss function when updating the parameters. Remember that the gradient gives us the slope of a function at that point. Without being able to go into detail, the process consists in chaining the derivatives of the loss of each hidden layer from the derivatives of the loss of its upper layer, incorporating its activation function in the calculation (that's why the activation functions must be derivable). In each of the iterations, once all the neurons have the value of the gradient of the loss function that corresponds to them, the values of the parameters are

updated in the opposite direction to that indicated by the gradient. The gradient, in fact, always points in the direction in which the value of the loss function increases. Therefore, if the negative of the gradient is used, we can get the direction in which we tend to reduce the loss function.

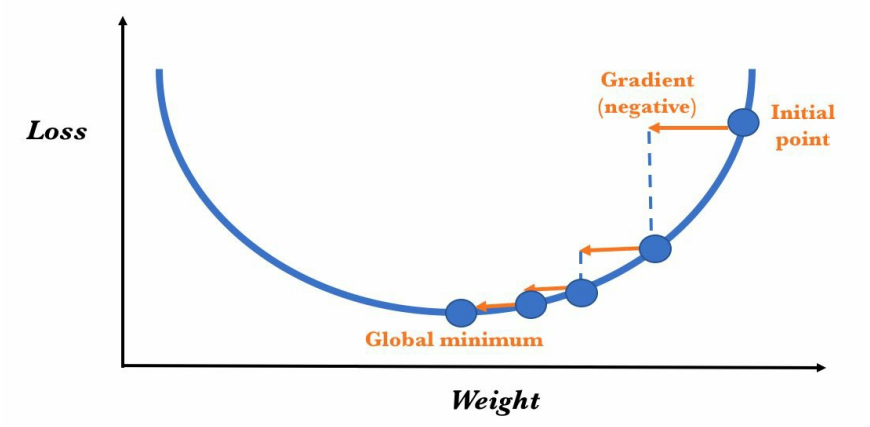
Let's see the process in a visual way assuming only one dimension: suppose that this line represents the values that the loss function takes for each possible parameter value and that the negative of the gradient is depicted by the arrow in the initial point:



To determine the next value for the parameter, the gradient descent algorithm modifies the value of the initial weight to go in the opposite way to the gradient (since it points in the direction in which the loss grows and we want to reduce it), adding a proportional amount to this. The magnitude of this change is determined by the value of the gradient and by a learning rate hyperparameter that we can specify (which we will present shortly). Therefore, conceptually, it is as if we follow the slope downhill until we reach a local minimum:



The gradient descent algorithm repeats this process getting closer and closer to the minimum until the value of the parameter reaches a point beyond which the loss function cannot decrease:



Stochastic Gradient Descent (SGD)

In the previous sections we have seen how the values of the parameters are adjusted but not in what frequency:

- After each entry example?
- After each round of the whole set of training examples (epoch)?

- After a sample of examples of the training set?

In the first case we speak of online learning, when the gradient is estimated from the observed loss for each example of the training; it is also when we talk about Stochastic Gradient Descent (SGD). The second is known as batch learning and is called Batch Gradient Descent. The literature indicates that, usually, better results can be obtained with online learning, but there are reasons that justify the use of batch learning because many optimization techniques only work with it.

But if the data is well distributed, a small subset of them should give us a pretty good approximation of the gradient. We may not get the best estimate, but it is faster and, given the fact that we are iterating, this approach is very useful. For this reason, the third aforementioned option known as mini-batch is often used. This option is usually as good as the online, but fewer calculations are required to update the parameters of the neural network. In addition, the simultaneous calculation of the gradient for many input examples can be done using matrix operations that are implemented very efficiently with GPU, as we have seen in chapter 1.

That's why, in reality, many applications use the stochastic gradient descent (SGD) with a mini-bach of several examples. To use all the data, what is done is to partition the data in several batches. Then we take the first batch, go through the network, calculate the gradient of its loss and update the parameters of the neural network; this would follow successively until the last batch. Now, in a single pass through all the input data, only a number of steps have been made equal to the number of batches.

SGD is very easy to implement in Keras. In the *compile()* method it is indicated that the optimizer is SGD (value *sgd* in the argument), and then all that must be done is to specify the batch size in the training process with the *fit()* method as follows :

```
model.fit(X_train, y_train, epochs=5, batch_size=100)
```

In this code example that uses the *fit()* method, we are dividing our data into batches of 100 with the *batch_size* argument. With the number of *epochs* we are indicating how many times we carry out this process on all the data. Later in this chapter, when we have already presented the usual optimizer parameters, we will return to these two arguments.

4 Model parameterization

If the reader at the end of the previous chapter has executed a model with the hyperparameters that we use there, I assume that the model's accuracy will have exceeded 90%. Are these results good? I think they are fantastic, because it means that the reader has already programmed and executed his first neural network with Keras. *Congratulations!*

Another thing is that there are other models that improve accuracy. And this depends on having a great knowledge and a lot of practice to handle well with the many hyperparameters that we can change. For example, with a simple change of the activation function of the first layer, passing from a *sigmoid* to a *relu* like the one shown below:

```
model.add(Dense(10, activation='relu', input_shape=(784,)))
```

We can get 2% more accuracy with more or less the same calculation time.

It is also possible to increase the number of *epochs*, add more neurons in a layer or add more layers. However, in these cases, the gains in accuracy have the side effect of increasing the execution time of the learning process. For example, if we add 512 nodes to the intermediate layer instead of 10 nodes:

```
model.add(Dense(512, activation='relu', input_shape=(784,)))
```

We can check with the *summary()* method that the number of parameters increases (it is a *fully connected*) and the execution time is significantly

higher, even reducing the number of epochs. With this model, the accuracy reaches 94%. And if we increase to 20 epochs, a 96% accuracy is achieved.

In short, an immense world of possibilities will be seen in more detail in the following chapters, but the reader can already realize that finding the best architecture with the best parameters and hyperparameters of activation functions requires some expertise and experience given the multiple possibilities that we have.

Parameters and hyperparameters

So far, for simplicity, we have not paid explicit attention to differentiating between parameters and hyperparameters, but I think the time has come. In general, we consider a parameter of the model as a configuration variable that is internal to the model and whose value can be estimated from the data. In contrast, by hyperparameter we refer to configuration variables that are external to the model itself and whose value in general cannot be estimated from the data, and are specified by the programmer to adjust the learning algorithms.

When I say that Deep Learning is more an art than a science, I mean that it takes a lot of experience and intuition to find the optimal values of these hyperparameters, which must be specified before starting the training process so that the models train better and more quickly. Given the introductory nature of the book we will not go into detail about all of them, but we have hyperparameters that are worth mentioning briefly, both at the structure and topology level of the neural network (number of layers, number of neurons, their activation functions, etc.) and at the learning algorithm level (learning rate, momentum, epochs, batch size, etc.).

Next, we will introduce some of them and the rest will appear in chapter 6 as we go into convolutional neural network.

Epochs

As we have already done, epochs tells us the number of times all the training data have passed through the neural network in the training process. A good clue is to increase the number of epochs until the accuracy metric with the validation data starts to decrease, even when the accuracy of the training data continues to increase (this is when we detect a potential overfitting).

Batch size

As we have said before, we can partition the training data in mini batches to pass them through the network. In Keras, the *batch_size* is the argument that indicates the size of these batches that will be used in the *fit()* method in an iteration of the training to update the gradient. The optimal size will depend on many factors, including the memory capacity of the computer that we use to do the calculations.

Learning rate

The gradient vector has a direction and a magnitude. Gradient descent algorithms multiply the magnitude of the gradient by a scalar known as learning rate (also sometimes called step size) to determine the next point.

In a more formal way, the backpropagation algorithm computes how the error changes with respect to each weight:

$$\frac{dError}{dw_{ij}}$$

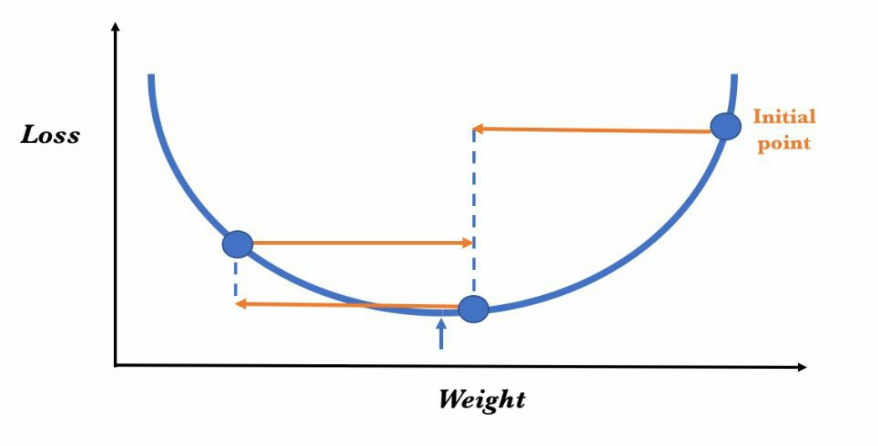
In order to update each weight of the network using a simple update rule:

$$w_{ij} = w_{ij} - \alpha \frac{dError}{dw_{ij}}$$

Where α is the learning rate.

For example, if the magnitude of the gradient is 1.5 and the learning rate is 0.01, then the gradient descent algorithm will select the next point at 0.015 from the previous point.

The proper value of this hyperparameter is very dependent on the problem in question, but in general, if this is too big, huge steps are being made, which could be good to go faster in the learning process. But in this case, we may skip the minimum and make it difficult for the learning process to stop because, when searching for the next point, it perpetually bounces randomly at the bottom of the "well". Visually we can see in this figure the effect that can occur, where the minimum value is never reached (indicated with a small arrow in the drawing):



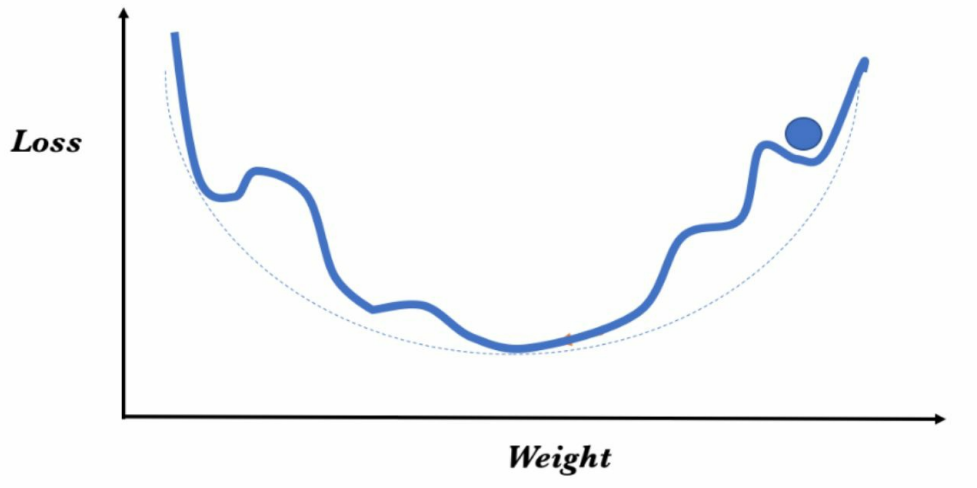
Contrarily, if the learning rate is small, small advances will be made, having a better chance of reaching a local minimum, but this can cause the learning process to be very slow. In general, a good rule is to decrease the learning rate if our learning model does not work. If we know that the gradient of the loss function is small, then it is safe to test that it compensates the gradient with the learning rate.

Learning rate decay

But the best learning rate in general is one that decreases as the model approaches a solution. To achieve this effect, we have another hyperparameter, the learning rate decay, which is used to decrease the learning rate as epochs go by to allow learning to advance faster at the beginning with larger learning rates. As progress is made, smaller and smaller adjustments are made to facilitate the convergence of the training process to the minimum of the loss function.

Momentum

In the visual example with which we have explained the descent gradient algorithm, to minimize the loss function we have the guarantee of finding the global minimum because there is no local minimum in which the optimization process can be stuck. However, in reality, the real cases are more complex and, visually, it is as if we could find several local minimums and the loss function had a form like the one in the following figure:



In this case, the optimizer can easily get stuck at a local minimum and the

algorithm may think that the global minimum has been reached, leading to suboptimal results. The reason is that the moment we get stuck, the gradient is zero and we can no longer get out of the local minimum strictly following the path of the gradient.

One way to solve this situation could be to restart the process from different random positions and, in this way, increase the probability of reaching the global minimum.

To avoid this situation, another solution that is generally used involves the *momentum* hyperparameter. In an intuitive way, we can see it as if, to move forward, it will take the weighted average of the previous steps to obtain a bit of impetus and overcome the "bumps" as a way of not getting stuck in local minima. If we consider that the average of the previous ones was better, perhaps it will allow us to make the jump.

But using the average has proved to be a very drastic solution because, perhaps in gradients of previous steps, it is much less relevant than just in the previous one. That is why we have chosen to weight the previous gradients, and the momentum is a constant between 0 and 1 that is used for this weighting. It has been shown that algorithms that use momentum work better in practice.

One variant is the *Nesterov momentum*, which is a slightly different version of the momentum update that has recently gained popularity and which basically slows down the gradient when it is close to the solution.

Initialization of parameter weights

The initialization of the parameters' weight is not exactly a hyperparameter, but it is as important as any of them and that is why we make a brief paragraph in this section. It is advisable to initialize the weights with small random values to break the symmetry between different neurons, if two

neurons have exactly the same weights they will always have the same gradient; that supposes that both have the same values in the subsequent iterations, so they will not be able to learn different characteristics.

Initializing the parameters randomly following a standard normal distribution is correct, but it can lead to possible problems of vanishing gradients (when the values of a gradient are too small and the model stops learning or takes too long due to that) or exploding gradients (when the algorithm assigns an exaggeratedly high importance to the weights).

In general, heuristics can be used taking into account the type of activation functions that our network has. It is outside the introductory level of this book to go into these details but, if the reader wants to go deeper, I suggest that you visit the CS231n course[130] website of Andrej Karpathy in Stanford, where you will obtain very valuable knowledge in this area exposed in a very didactic way.

Hyperparameters and optimizers in Keras

How can we specify these hyperparameters? Recall that the optimizer is one of the arguments that are required in the *compile()* method of the model. So far we have called them by their name (with a simple strings that identifies them), but Keras also allows to pass an instance of the optimizer class as an argument with the specification of some hyperparameters.

For example, the *stochastic gradient descent* optimizer allows the use of the *momentum*, *learning rate decay* and *Nesterov momentum* hyperparameters:

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

The values indicated in the arguments of the previous method are those taken by default and whose range can be:

- lr: float >= 0. (learning rate)
- momentum: float >= 0
- decay: float >= 0 (learning rate decay).
- nesterov: boolean (which indicates whether or not to use Nesterov momentum).

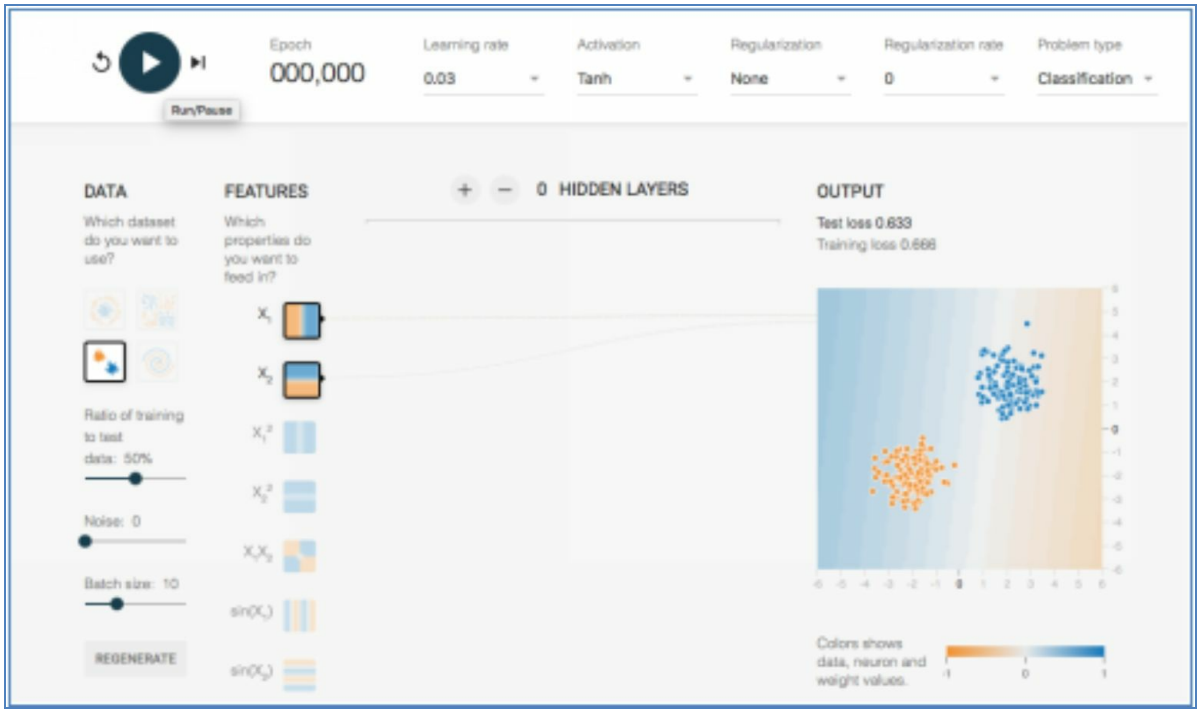
As we have said, there are several optimizers in Keras that the reader can explore on their documentation page[131].

Get started with Deep Learning hyparameters

In this chapter, we propose to test what we learned in the previous one with an interactive tool called TensorFlow Playground[132]. This tool can help the reader to enter details of a neural network and put into practice some of the concepts presented here without having to enter the mathematics behind it.

TensorFlow Playground

TensorFlow Playground is an interactive visualization web application written in JavaScript that allows us to simulate simple neural networks that run in our browser and see the results in real time:



With this tool we can experiment with different hyperparameters and see their behavior. In fact, the flexibility provided by the hyperparameters in neural networks is one of its virtues and at the same time one of its drawbacks for those who start on the subject: there are many of them to adjust!

2 A binary classification problem

To begin to understand how the tool works we can use the first example of perceptron presented in this book in chapter 2, a simple classification problem.

To start with this example, we chose the dataset indicated in the "DATA" section shown in the previous figure, and then click on the "Play" button. Now we can see how TensorFlow Playground solves this particular problem. The line between the blue and orange area begins to move slowly. You can press the "Reset" button and re-test it several times to see how the line moves with different initial values.

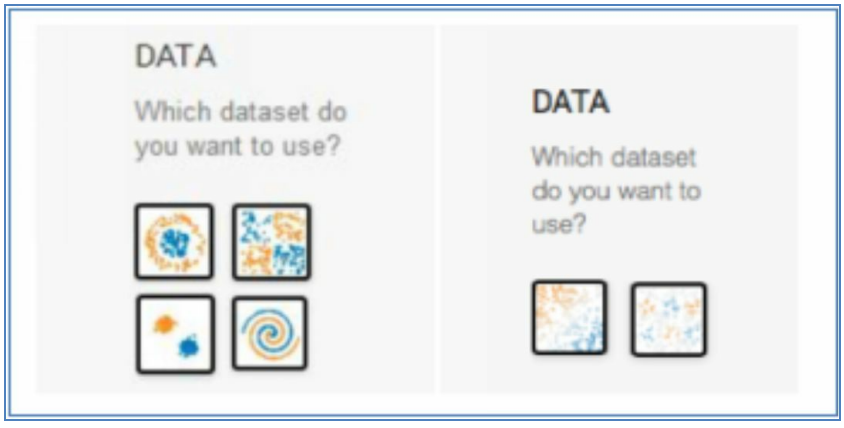
In this case, the application tries to find the best values of the parameters that allow classifying these points correctly. If the cursor is put over the arcs, the reader will see that the value that has been assigned to each parameter appears (and it even allows us to edit it):

Remember that this weight dictates the importance of this relationship in the neuron when multiplying it by the input value.

After this first contact, we will present a little the tool that will allow us to understand how a neural network behaves. In the upper part of the menu, we basically find hyperparameters, some of which we already commented in the previous chapter: Epoch, Learning rate, Activation, Regularization rate, and Problem type. All of them are drop-down menus in which we can choose the

value of these hyperparameters.

In the “Problem type” tab, the platform allows us to specify two types of problems: Regression (continuous problem) and Classification. In total, there are four types of data that we can choose for classification and two types for regression:



The blue and orange dots form the dataset. The orange points have the value -1 and the blue points the value +1. On the left side, below the data type, there are different parameters that we can modify to tune our input data.

Using the tab "Ratio of training to test data" we can control the percentage of data that is assigned to the training set (if we modify it we see how the points that appear in the "OUTPUT" on the right side of the screen are changed interactively). The noise level of the data can also be defined and controlled by the "Noise" field; the data pattern becomes more irregular as the noise increases. As we can experience, when the noise is zero, the problem data are clearly distinguished in their regions. However, when reaching more than 50, we can see that the blue dots and the orange dots are mixed, so it is very difficult to classify them.

With "Batch size", as its name suggests, we can determine the amount of data that will be used for each training batch.

Then, in the next column, we can make the selection of features. I propose that we use "X1" and "X2" among the many available to us: "X1" is a value on the horizontal axis, and "X2" is the value on the vertical axis.

The topology of the neuronal network can be defined in the following column. We can have up to six hidden layers (by adding hidden layers, by clicking on the "+" sign) and we can have up to eight neurons per hidden layer (by clicking on the "+" sign of the corresponding layer):

Finally, remember that when training the neural network we want to minimize the "Training loss" and then compare that with the test data the "Test loss" is also minimized. The changes of both metrics in each epoch are shown interactively in the upper right part of the screen, in a small graph where, if the loss is reduced, the curve goes downwards. The loss test is

painted in black, and the training loss is painted in gray.

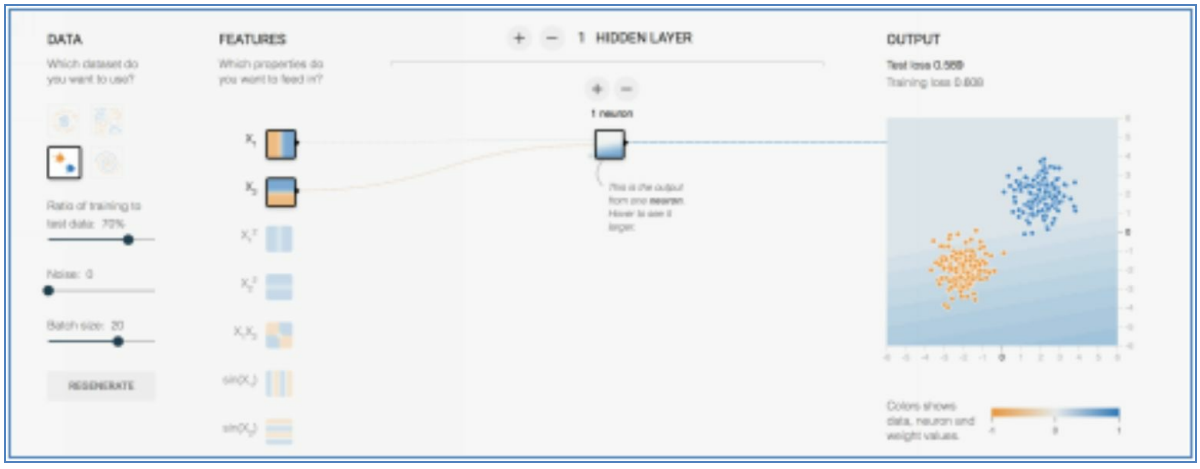
Classification with a single neuron

Now that we know a bit more about this tool, let's return to the first classification example that separates the data into two groups (clusters).

I propose that we modify some parameters to practice with the tool before moving forward. For example, we can modify some parameters with a learning rate of 0.03 and a ReLU activation function (the regularization is not going to be used since it is outside the scope of this book).

We maintain the problem as classification, and I propose that we put the "ratio of training-to-test" to 50% of the data and that we also keep the "noise" parameter to zero to facilitate the visualization of the solution (although I suggest that later you practice with it on your own). We can leave the "batch size" at 10.

And, as before, we will use "X1" and "X2" for the input. I suggest starting with a single hidden layer with a single neuron. We can achieve this by using the "-" or "+" buttons:



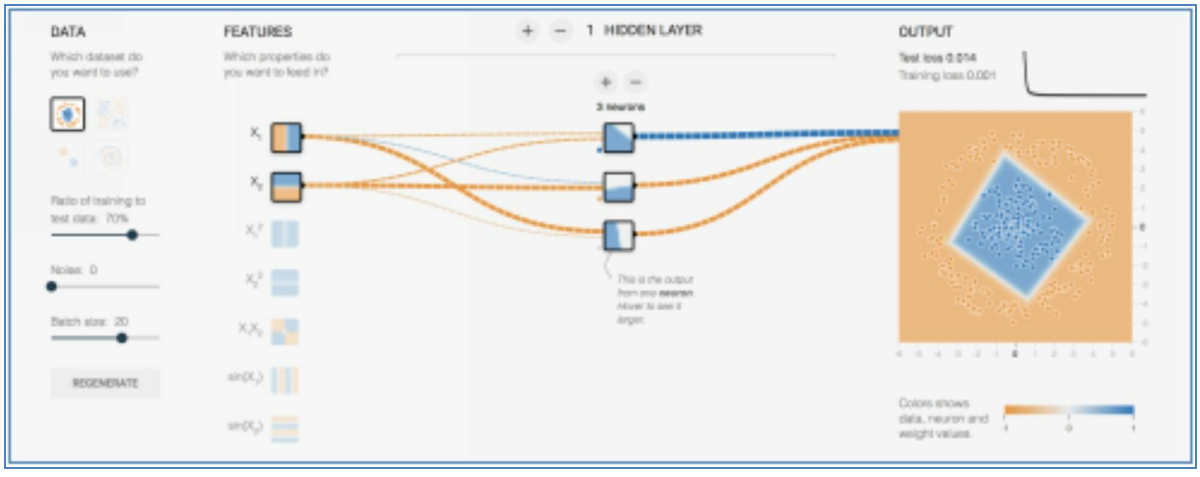
In the upper right, we see that the initial values of the "Test loss" and "Training loss" are high (the reader can get different values since the initial values are generated in a random way). But after pressing the "play" button, it can be seen that both the "Training loss" and the "Test loss" converge at very low ratios and remain the same. Moreover, in this case, both lines, black and gray, perfectly overlap.

Classification with more than one neuron

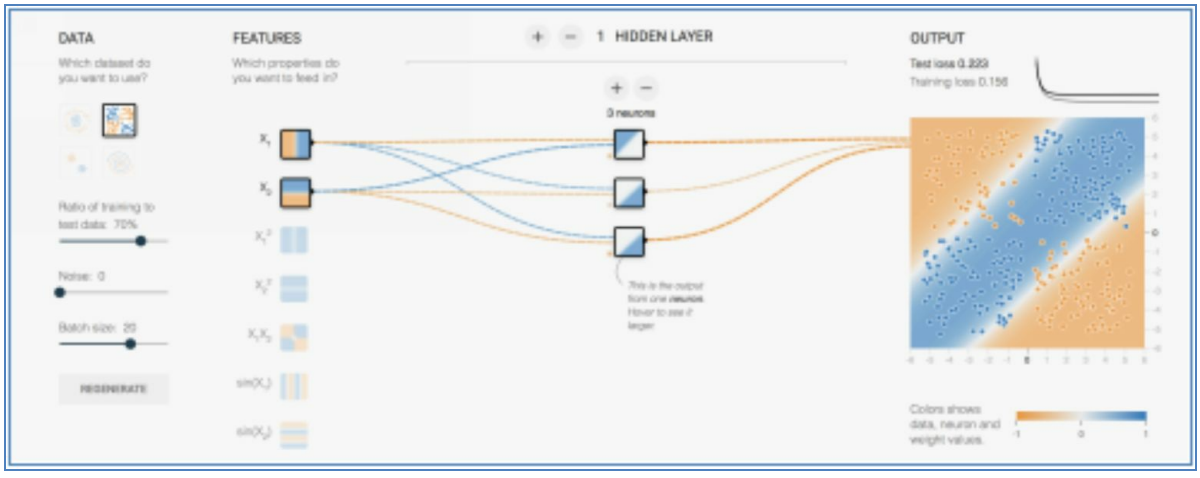
Let's choose another set of starting data like the one in the attached figure:

We now want to separate the two datasets: the orange ones must be classified in one group and the blue ones in another. But the problem is that in this case they will have a circular shape where the orange points will be in the outer circle and the blue points will be inside. Now, these points cannot be separated with a single line as before. If we train with a hidden layer that has a single neuron as the model of the previous classification, the classification will fail in this case.

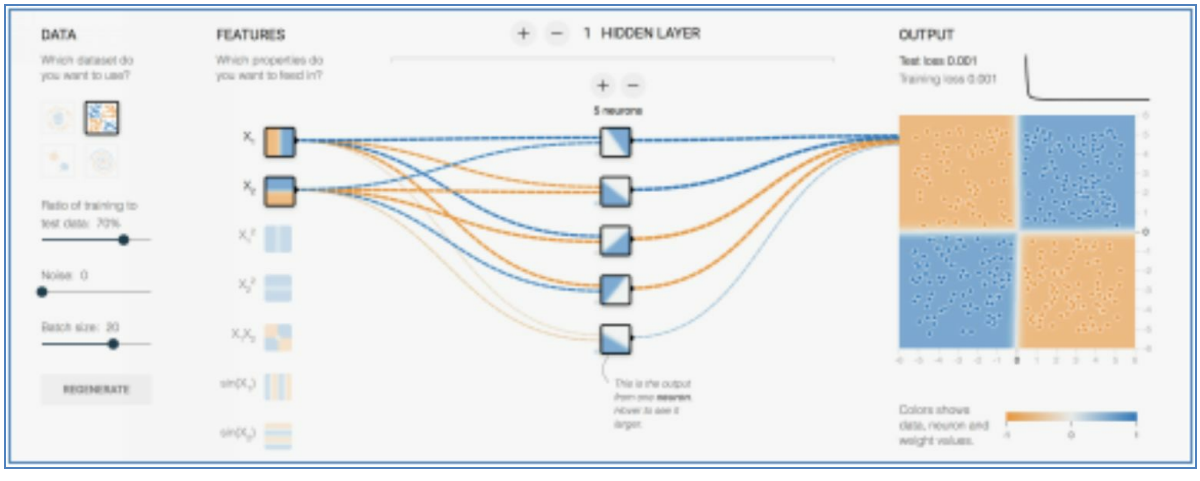
I suggest that we test with multiple neurons in the hidden layer. For example, try two neurons: you will see that you have not tuned enough yet. I propose that you then try with three. You will see that, in the end, you can get a much better training and test loss:



Let's go for another of the datasets on the left, that where the data is divided into four different square zones. Now, this problem cannot be solved with the previous network, but I propose that you try it:

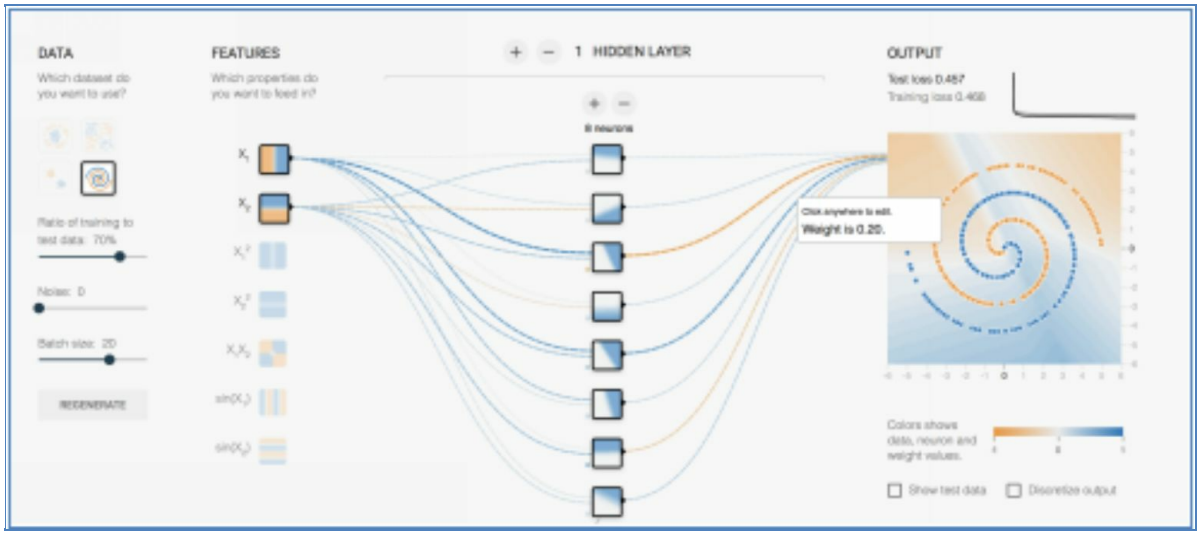


As can be seen, we are not able to get a good classification (although in some cases it could happen that with only 3 neurons it works since the initialization is random, but if you do several tests you will see that it is not achieved in general). However, if we have 5 neurons as in the following figure, the reader can see how this neural network gets a good classification for this case:

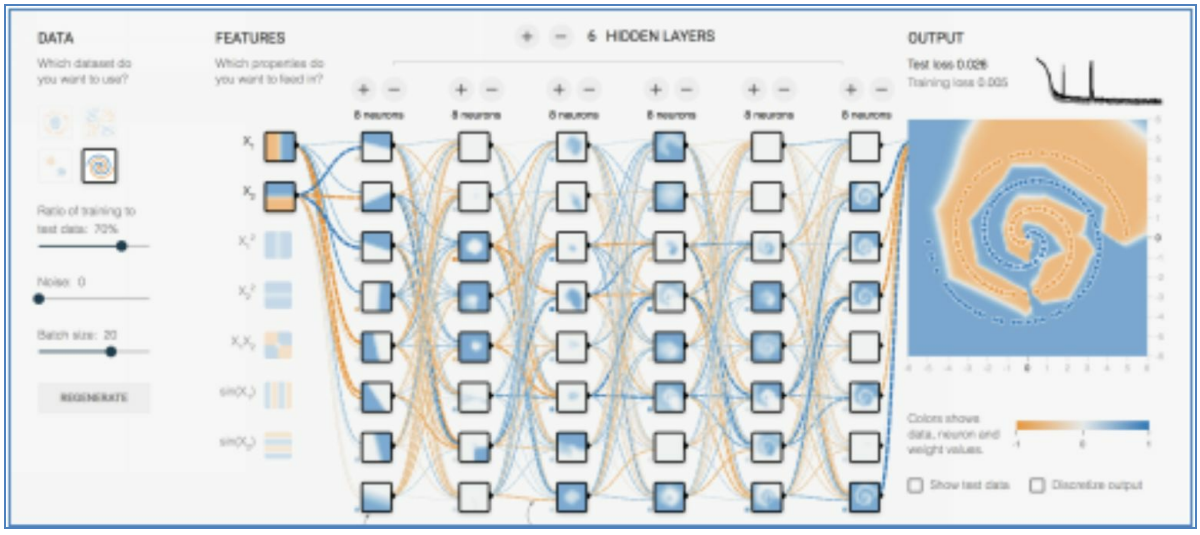


Classification with several layers

Now we will try to classify the dataset with the most complex pattern that we have in this tool. The swirling structure of the orange and blue data points is a challenging problem. If we rely on the previous network, we see that not even having 8 neurons, the maximum that the tool leaves us, we get a good classification result:

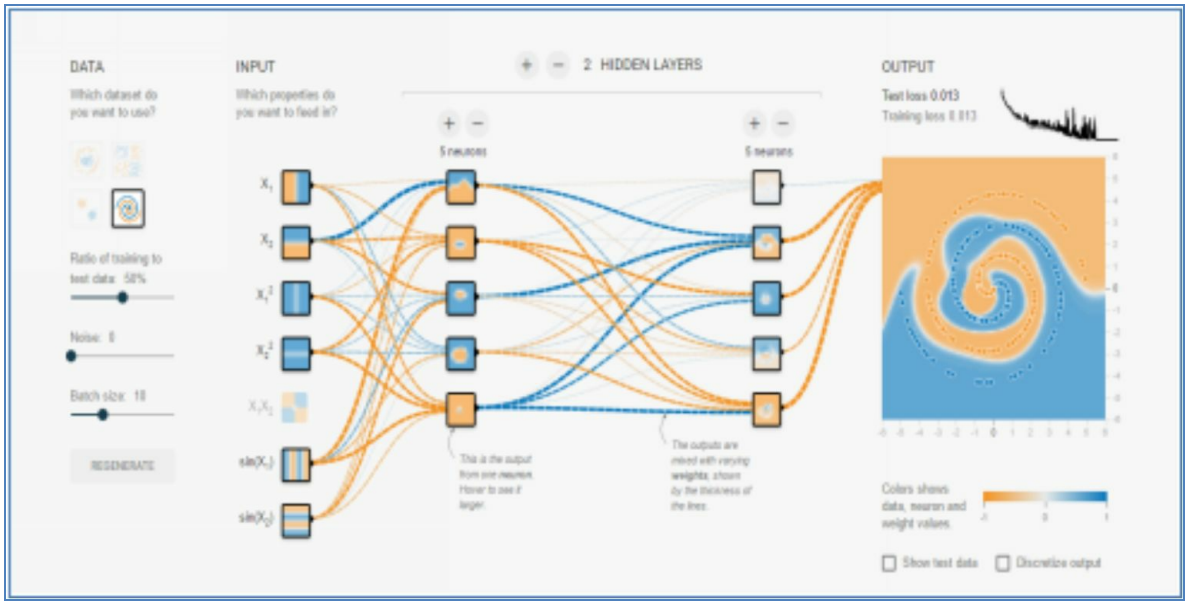


If the reader has tried it, in this case I suppose that you will get a few good values for the test loss. The time has come to put more layers; I assure you that, if you use all the layers the tool allows, you will get it:



But you will see that, as it is obvious, the process of learning the parameters takes a long time.

Actually, with fewer layers or neurons you can get good results; I challenge you to play a little on your own, also changing for example the activation functions to get a simpler model. You can also consider testing any other parameter.



This tool only considers dense neural networks; later we will see that the

convolutional neural networks (and the recurrent neural networks that we will not deal with in this book) present more complex dilemmas. But only with these dense networks can we see that one of the most difficult hyperparameters to adjust is to decide how many layers the model has and how many neurons each of these layers has.

Using very few neurons in the hidden layers will result in what is called underfitting, a lack of fit of the model because there are very few neurons in the hidden layers to properly detect the signals in a complicated dataset.

On the other hand, using too many neurons in the hidden layers can cause several problems. First, it can produce overfitting, which occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all the neurons in the hidden layers. But on the other hand, a large number of neurons in the hidden layers can increase the time needed to train the network to the point that it is impossible to properly train the neural network in the necessary time.

Obviously, we must reach a compromise between too many and very few neurons in the hidden layers and that is why I have already commented that we are facing a challenge that requires more art than science.

Convolutional Neural Networks

At this point, we are ready to deal with another type of neural networks, the so-called convolutional neuronal networks, widely used in computer vision tasks. These networks are composed of an input layer, an output layer and several hidden layers, some of which are convolutional, hence its name.

In this chapter we will present a specific case that we will follow step by step to understand the basic concepts of this type of networks. Specifically, together with the reader, we will program a convolutional neural network to solve the same MNIST digit recognition problem seen above.

Introduction to convolutional neural networks

A convolutional neuronal network (with the acronyms CNNs or ConvNets) is a concrete case of Deep Learning neural networks, which were already used at the end of the 90s but which in recent years have become enormously popular when achieving very impressive results in the recognition of image, deeply impacting the area of computer vision.

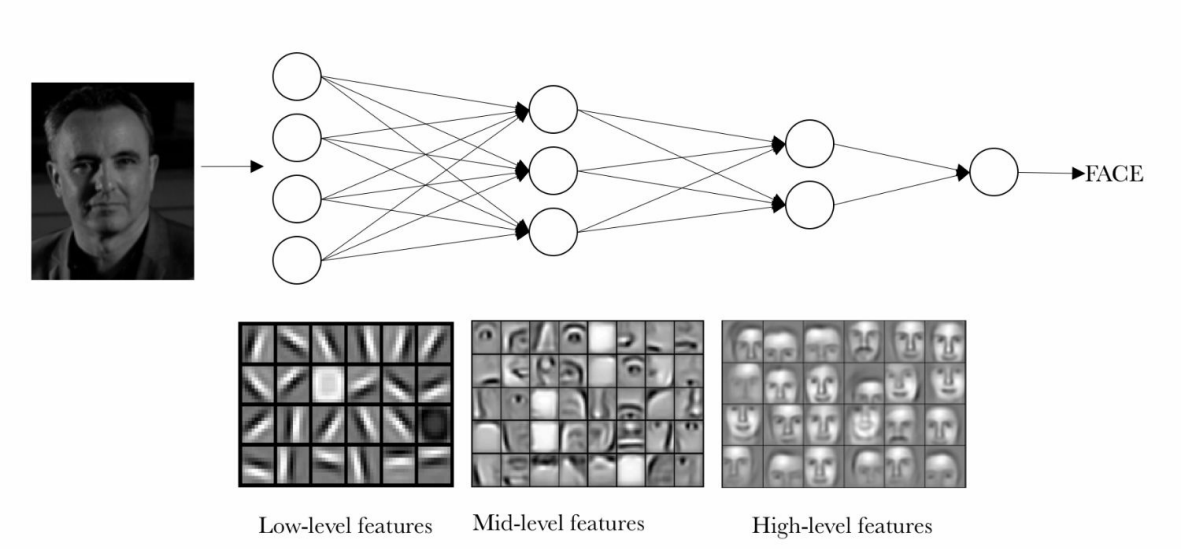
The convolutional neural networks are very similar to the neural networks of the previous chapters: they are formed by neurons that have parameters in the form of weights and biases that can be learned. But a differential feature of the CNN is that they make the explicit assumption that the entries are images, which allows us to encode certain properties in the architecture to recognize specific elements in the images.

To get an intuitive idea of how these neural networks work, let's think about how we recognize things. For example, if we see a face, we recognize it because it has ears, eyes, a nose, hair, etc. Then, to decide if something is a face, we do it as if we had some mental boxes of verification of the characteristics that we are marking. Sometimes a face may not have an ear because it is covered by hair, but we also classify it with a certain probability as face because we see the eyes, nose and mouth. Actually, we can see it as a classifier equivalent to the one presented in chapter 2, which predicts a probability that the input image is a face or no face.

But in reality, we must first know what an ear or a nose is like to know if they are in an image; that is, we must previously identify lines, edges, textures or shapes that are similar to those containing the ears or noses we have seen before. And this is what the layers of a convolutional neuronal network are entrusted to do.

But identifying these elements is not enough to be able to say that something is a face. We also must be able to identify how the parts of a face meet each

other, relative sizes, etc.; otherwise, the face would not resemble what we are used to. Visually, an intuitive idea of what layers learn is often presented with this example from this reference article by Andrew Ng's group[133].



The idea that we want to give with this visual example is that, in reality, in a convolutional neural network each layer is learning different levels of abstraction. The reader can imagine that, with networks with many layers, it is possible to identify more complex structures in the input data.

2 Basic components of a convolutional neural network neuronal

Now that we have an intuitive vision of how convolutional neural networks classify an image, we will present an example of recognition of MNIST digits and from it we will introduce the two layers that define convolutional networks that can be expressed as groups of specialized neurons in two operations: convolution and pooling.

The convolution operation

The fundamental difference between a densely connected layer and a specialized layer in the convolution operation, which we will call the convolutional layer, is that the dense layer learns global patterns in its global input space, while the convolutional layers learn local patterns in small windows of two dimensions.

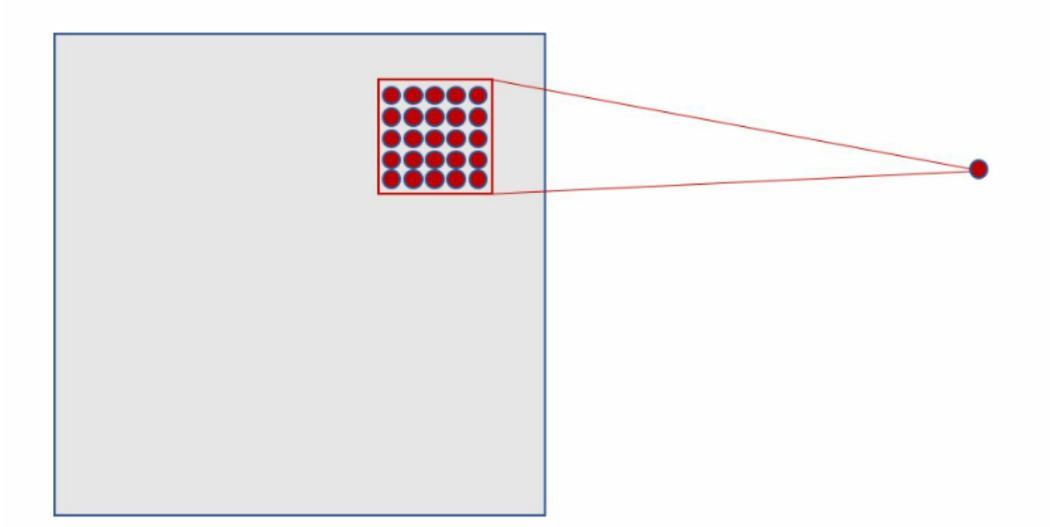
In an intuitive way, we could say that the main purpose of a convolutional layer is to detect features or visual features in images such as edges, lines, color drops, etc. This is a very interesting property because, once it has learned a characteristic at a specific point in the image, it can recognize it later in any part of it. Instead, in a densely connected neural network it has to learn the pattern again if it appears in a new location of the image.

Another important feature is that convolutional layers can learn spatial hierarchies of patterns by preserving spatial relationships. For example, a first convolutional layer can learn basic elements such as edges, and a second convolutional layer can learn patterns composed of basic elements learned in the previous layer. And so on until it learns very complex patterns. This allows convolutional neural networks to efficiently learn increasingly complex and abstract visual concepts.

In general, the convolutions layers operate on 3D tensors, called feature maps, with two spatial axes of height and width, as well as a channel axis also called depth. For an RGB color image, the dimension of the depth axis is 3, because the image has three channels: red, green and blue. For a black and white image, such as the MNIST digits, the depth axis dimension is 1 (gray level).

In the case of MNIST, as input to our neural network we can think of a space of two-dimensional neurons 28×28 (height = 28, width = 28, depth = 1). A first layer of hidden neurons connected to the neurons of the input layer that we have discussed will perform the convolutional operations that we have just described. But as we have explained, not all input neurons are connected with all the neurons of this first level of hidden neurons, as in the case of densely connected neural networks; it is only done by small localized areas of the space of input neurons that store the pixels of the image.

The explained, visually, could be represented as:

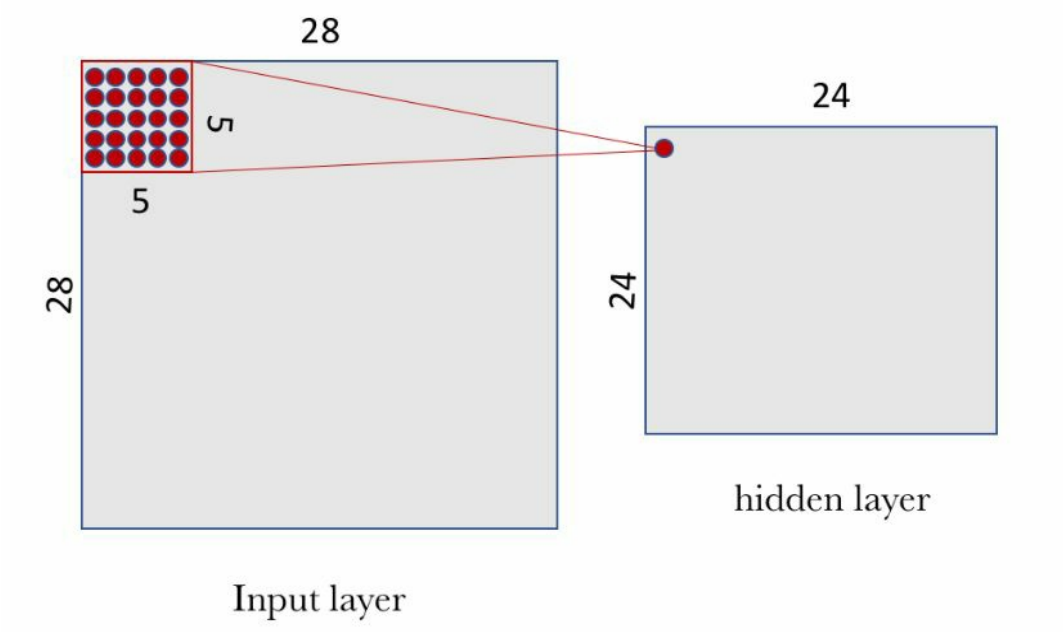


In the case of our previous example, each neuron of our hidden layer will be connected to a small region of 5×5 neurons (i.e. 25 neurons) of the input layer (28×28). Intuitively, we can think of a 5×5 size window that slides along the entire 28×28 neuron layer of input that contains the image. For each position of the window there is a neuron in the hidden layer that processes

this information.

Visually, we start with the window in the top left corner of the image, and this gives the necessary information to the first neuron of the hidden layer. Then, we slide the window one position to the right to "connect" the 5×5 neurons of the input layer included in this window with the second neuron of the hidden layer. And so, successively, we go through the entire space of the input layer, from left to right and top to bottom.

Analyzing a little bit the concrete case we have proposed, we note that, if we have an input of 28×28 pixels and a window of 5×5, this defines a space of 24×24 neurons in the first hidden layer because we can only move the window 23 neurons to the right and 23 neurons to the bottom before hitting the right (or bottom) border of the input image.



We would like to point out to the reader that the assumption we have made is that the window moves forward 1 pixel away, both horizontally and vertically when a new row starts. Therefore, in each step, the new window overlaps the previous one except in this line of pixels that we have advanced. But, as we will see in the next section, in convolutional neural networks, different

lengths of advance steps can be used (the parameter called stride). In convolutional neural networks you can also apply a technique of filling zeros around the margin of the image to improve the sweep that is done with the sliding window. The parameter to define this technique is called "padding", which we will also present in more detail in the next section, with which you can specify the size of this padding.

In our case of study, and following the formalism previously presented, to "connect" each neuron of the hidden layer with the 25 corresponding neurons of the input layer we will use a bias value *b* and a *W*-weights matrix of size 5×5 that we will call filter (or kernel). The value of each point of the hidden layer corresponds to the scalar product between the filter and the handful of 25 neurons (5×5) of the input layer.

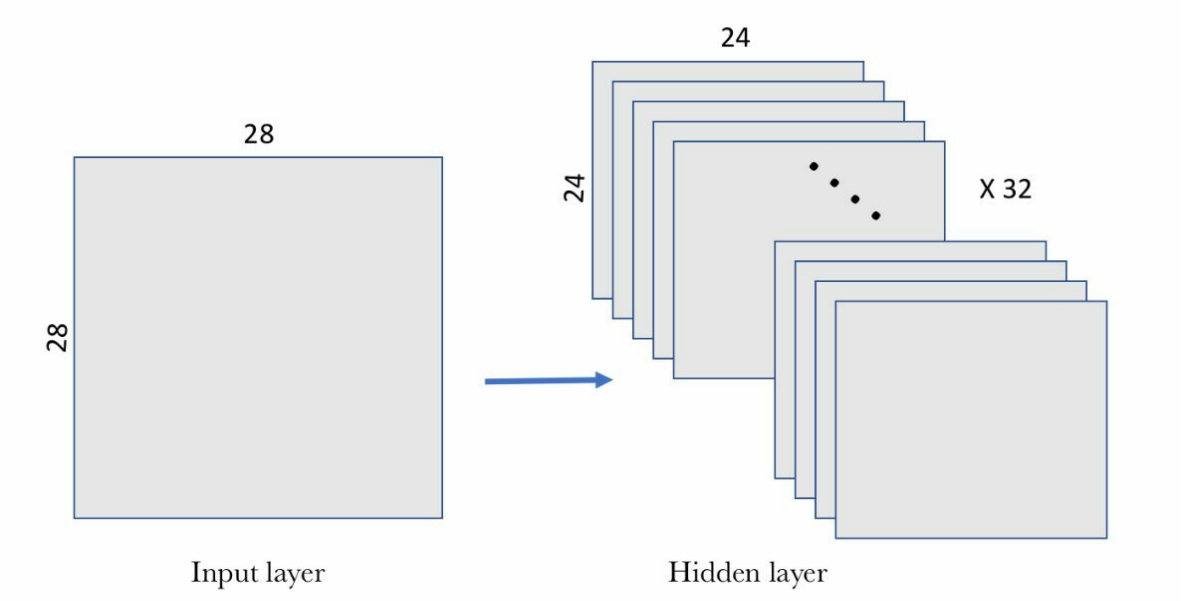
However, the particular and very important thing about convolutional networks is that we use the same filter (the same *W* matrix of weights and the same *b* bias) for all the neurons in the hidden layer: in our case for the 24×24 neurons (576 neurons in total) of the first layer. The reader can see in this particular case that this sharing drastically reduces the number of parameters that a neural network would have if we did not do it: it goes from 14,400 parameters that would have to be adjusted (5×5 × 24×24) to 25 (5×5) parameters plus biases *b*.

This shared *W* matrix together with the *b* bias, which we have already said we call a filter in this context of convolutional networks, is similar to the filters we use to retouch images, which in our case are used to look for local characteristics in small groups of entries. I recommend looking at the examples found in the GIMP image editor manual[134] to get a visual and very intuitive idea of how a convolution process works.

But a filter defined by a matrix *W* and a bias *b* only allows detecting a specific characteristic in an image; therefore, in order to perform image recognition, it is proposed to use several filters at the same time, one for each

characteristic that we want to detect. That is why a complete convolutional layer in a convolutional neuronal network includes several filters.

A usual way to visually represent this convolutional layer is shown in the following figure, where the level of hidden layers is composed of several filters. In our example we propose 32 filters, where each filter is defined with a *W* matrix of 5×5 and a bias *b*.



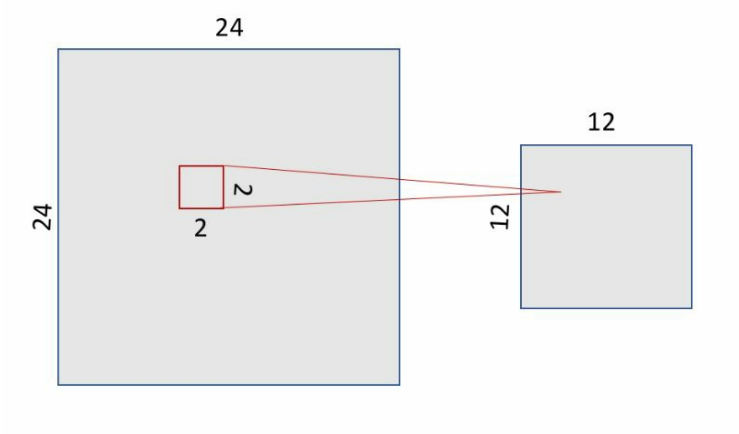
In this example, the first convolutional layer receives a size input tensor (28, 28, 1) and generates a size output (24, 24, 32), a 3D tensor containing the 32 outputs of 24×24 pixel result of computing the 32 filters on the input.

The pooling operation

In addition to the convolutional layers that we have just described, convolutional neural networks accompany the convolution layer with pooling layers, which are usually applied immediately after the convolutional layers. A first approach to understand what these layers are for is to see that the

pooling layers simplify the information collected by the convolutional layer and create a condensed version of the information contained in them.

In our MNIST example, we are going to choose a 2×2 window of the convolutional layer and we are going to synthesize the information in a point in the pooling layer. Visually, it can be expressed as follows:

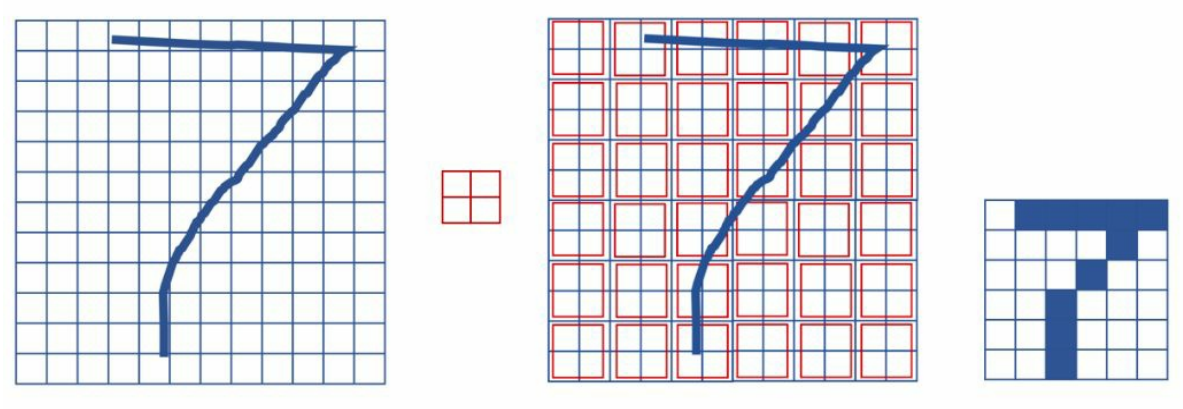


There are several ways to condense the information, but a usual one, which we will use in our example, is known as max-pooling, which as a value keeps the maximum value of those that were in the 2×2 input window in our case. In this case we divide by 4 the size of the output of the pooling layer, leaving an image of 12×12.

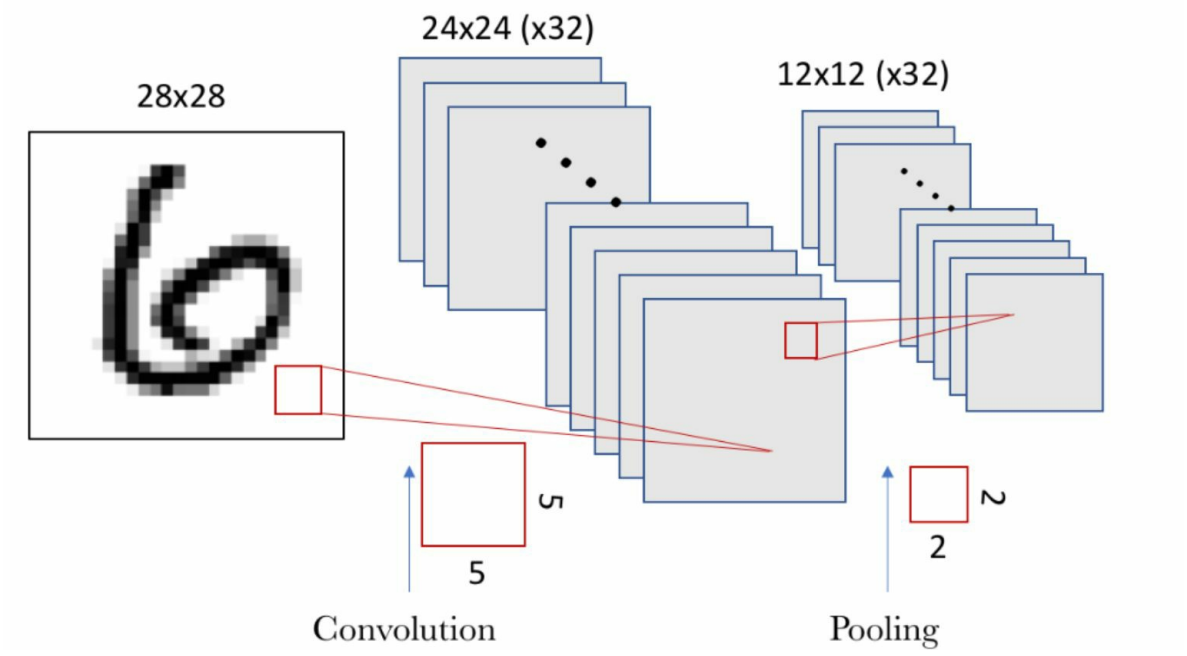
Average-pooling can also be used instead of max-pooling, where each group of entry points is transformed into the average value of the group of points instead of its maximum value. But in general, max-pooling tends to work better than alternative solutions.

It is interesting to note that with the transformation of pooling we maintain the spatial relationship. To see it visually, take the following example of a 12×12 matrix where we have represented a "7" (Let's imagine that the pixels where we pass over contain 1 and the rest 0; we have not added it to the drawing to simplify it). If we apply a max-pooling operation with a 2×2

window (we represent it in the central matrix that divides the space in a mosaic with regions of the size of the window), we obtain a 6×6 matrix where an equivalent representation of 7 is maintained (in the figure on the right where the zeros are marked in white and the points with value 1 in black):



As mentioned above, the convolutional layer hosts more than one filter and, therefore, as we apply the max-pooling to each of them separately, the pooling layer will contain as many pooling filters as there are convolutional filters:



The result is, since we had a space of 24×24 neurons in each convolutional filter, after doing the pooling we have 12×12 neurons which corresponds to the 12×12 regions (of size 2×2 each region) that appear when dividing the filter space.

3 Implementation of a basic model in Keras

Let's see how this example of convolutional neuronal network can be programmed using Keras. As we have said, there are several values to be specified in order to parameterize the convolution and pooling stages. In our case, we will use a simplified model with a stride of 1 in each dimension (size of the step with which the window slides) and a padding of 0 (filling with zeros around the image). Both hyperparameters will be presented below. The pooling will be a max-pooling as described above with a 2×2 window.

Basic architecture of a convolutional neuronal network

Let's move on to implement our first convolutional neuronal network, which will consist of a convolution followed by a max-pooling. In our case, we will have 32 filters using a 5×5 window for the convolutional layer and a 2×2 window for the pooling layer. We will use the ReLU activation function. In this case, we are configuring a convolutional neural network to process an input tensor of size (28, 28, 1), which is the size of the MNIST images (the third parameter is the color channel which in our case is depth 1), and we specify it by means of the value of the argument *input_shape = (28, 28,1)* in our first layer:

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32,(5,5),activation='relu',input_shape=(28, 28,1)))
model.add(layers.MaxPooling2D((2, 2)))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
conv2d_1 (Conv2D)           (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
=================================================================
Total params: 832
Trainable params: 832
Non-trainable params: 0
```

The number of parameters of the conv2D layer corresponds to the weight matrix W of 5×5 and a *b* bias for each of the filters is 832 parameters (32 × (25 + 1)). Max-pooling does not require parameters since it is a mathematical operation to find the maximum.

A simple model

And in order to build a "deep" neural network, we can stack several layers like the one built in the previous section. To show the reader how to do it in our example, we will create a second group of layers that will have 64 filters with a 5×5 window in the convolutional layer and a 2×2 window in the pooling layer. In this case, the number of input channels will take the value of the 32 features that we have obtained from the previous layer, although, as we have seen previously, it is not necessary to specify it because Keras deduces it:

```
model = models.Sequential()
model.add(layers.Conv2D(32,(5,5),activation='relu',input_shape=(28,28,1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

If the architecture of the model is shown with the *summary()* method, we can see:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 64)          51264
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0
=================================================================
Total params: 52,096
Trainable params: 52,096
Non-trainable params: 0
_________________________________________________________________
```

In this case, the size of the resulting second convolution layer is 8×8 since we now start from an input space of 12×12×32 and a sliding window of 5×5, taking into account that it has a stride of 1. The number of parameters 51,264 corresponds to the fact that the second layer will have 64 filters (as we have specified in the argument), with 801 parameters each (1 corresponds to the bias, and a W matrix of 5×5 for each of the 32 entries). That means ((5 × 5×32) +1) ×64 = 51264.

The reader can see that the output of the *Conv2D* and *MaxPooling2D* layers is a 3D form tensor (height, width, channels). The width and height dimensions tend to be reduced as we enter the hidden layers of the neural network. The number of kernels is controlled through the first argument passed to the *Conv2D* layer (usually size 32 or 64).

The next step, now that we have 64 4x4 filters, is to add a densely connected layer, which will serve to feed a final layer of softmax like the one introduced in chapter 2 to do the classification:

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

In this example, we have to adjust the tensors to the input of the dense layer like the softmax, which is a 1D tensor, while the output of the previous one is a 3D tensor. That's why we have to first flatten the 3D tensor to one of 1D. Our output (4,4,64) must be flattened to a vector of (1024) before applying the Softmax.

In this case, the number of parameters of the softmax layer is 10 × 1024 + 10, with an output of a vector of 10 as in the example in chapter 2:

```
model = models.Sequential()

model.add(layers.Conv2D(32,(5,5),activation='relu', input_shape=(28,28,1)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

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

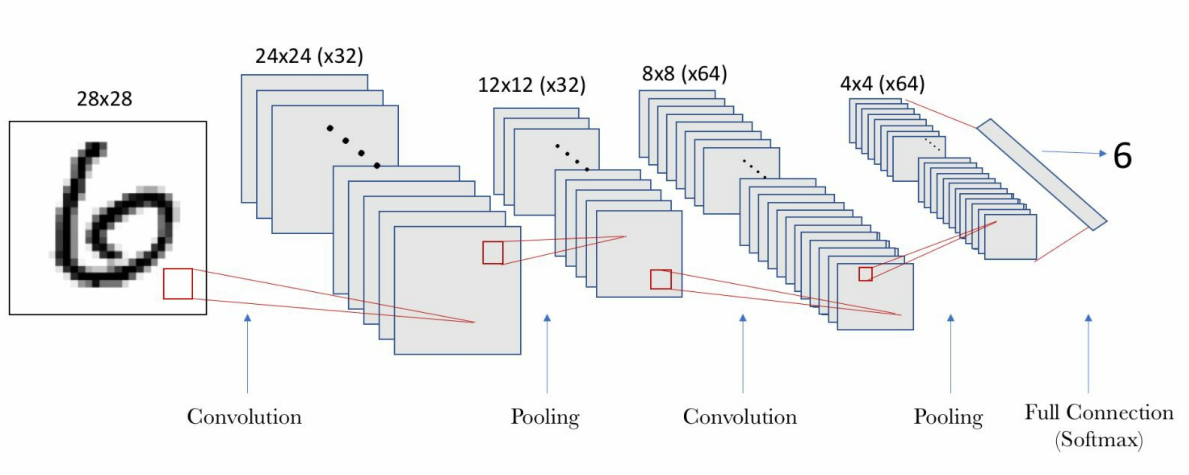
With the *summary()* method, we can see this information about the parameters of each layer and shape of the output tensors of each layer:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 64)          51264
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
```

```
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                10250
=================================================================
Total params: 62,346
Trainable params: 62,346
Non-trainable params: 0
_________________________________________________________________
```

Observing this summary, it is easily appreciated that in the convolutional layers is where more memory is required and, therefore, more computation to store the data. In contrast, in the densely connected layer of softmax, little memory space is needed but, in comparison, the model requires numerous parameters which must be learned. It is important to be aware of the sizes of the data and the parameters because, when we have models based on convolutional neural networks, they have many layers, as we will see later, and these values can shoot exponentially.

A more visual representation of the above information is shown in the following figure, where we see a graphic representation of the shape of the tensors that are passed between layers and their connections:



Training and evaluation of the model

Once the neural network model is defined, we are ready to train the model, that is, adjust the parameters of all the convolutional layers. From here, to know how well our model does, we must do the same as we did in the example of chapter 3. For this reason, and to avoid repetitions, we will reuse the code already presented above:

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) =
                        mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)




model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(train_images, train_labels,
          batch_size=100,
          epochs=5,
          verbose=1)
```

```
test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.9704
```

As in the previous cases, the code can be found in the GitHub of the book and it can be verified that this code offers an accuracy of approximately 97%.

If the reader has executed the code in a computer with only a CPU, it will have been noticed that, this time, the training of the network has taken a lot longer than the previous example, even with only 5 epochs. Can you imagine how long a network of many more layers, epochs or images could take? From here, as we already discussed in the introduction of the book, we need to train neuronal networks for real cases with more computing resources, such as GPUs.

The arguments of the fit method

The reader will have noticed that in this example we have not separated a part of the data for the validation of the model as we indicated in a previous section, which would be the ones that would be passed in the *validation_data* argument of the *fit()* method. How is it possible?

As we have seen in other cases, Keras takes many values by default, and one of them is this. Actually, if the *validation_data* argument is not specified, Keras uses the *validation_split* argument, which is an integer between 0 and 1 that specifies the fraction of the training data that should be considered as validation data (argument that we have not indicated in this example).

For example, a value of 0.2 implies that 20% of the data indicated in the Numpy arrays will be separated in the first two arguments of the *fit()* method

and will not be included in the training, being used only to evaluate the loss or any other metric at the end of each epoch. Its default value is 0.0. That is, if none of these arguments is specified, the validation process is not performed at the end of each epoch.

Actually, it does not make much sense not to do it, because the hyperparameters that we pass as arguments to the methods are very important and precisely the use of the validation data is crucial to find its best value. But in this introductory book we have thought it convenient not to go into these details and here is the simplification.

Also, this example serves to highlight that Keras has most of the hyperparameters initialized by default, in such a way that it facilitates beginners to start in the implementation of a neural network.

1 Hyperparameters of the convolutional layer

The main hyperparameters of the convolutional neural networks not seen until now are: the size of the filter window, the number of filters, the stride and padding.

Size and number of filters

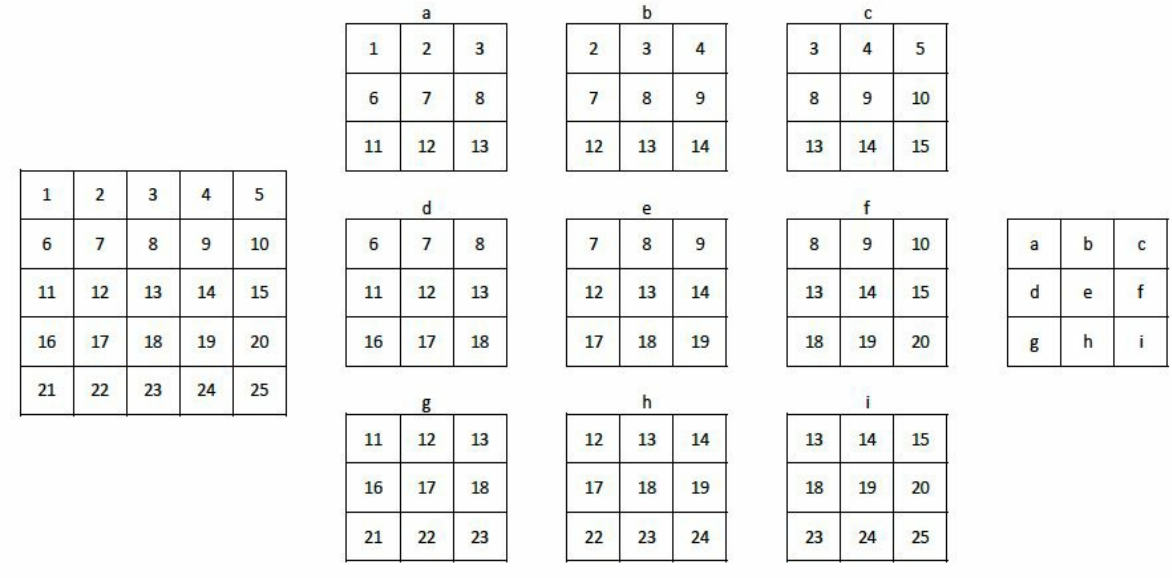
The size of the window (*window_height* × *window_width*) that holds information from spatially close pixels is usually 3×3 or 5×5. The number of filters that tells us the number of characteristics that we want to handle (output_depth) is usually 32 or 64. In the Conv2D layers of Keras, these hyperparameters are what we pass as arguments in this order:

```
Conv2D(output_depth, (window_height, window_width))
```

Padding

To explain the concept of padding let's use an example. Let's suppose an image with 5×5 pixels. If we choose a 3×3 window to perform the convolution, we see that the tensor resulting from the operation is of size 3×3. That is, it shrinks a bit: exactly two pixels for each dimension, in this case. In the following figure it is visually displayed. Suppose that the figure on the left is the 5×5 image. In it, we have numbered the pixels to make it easier to see how the 3×3 drop moves to calculate the elements of the filter. In the center, it is represented how the 3×3 window has moved through the image, 2 positions to the right and two positions to the bottom. The result of applying the convolution operation returns the filter that we have represented on the left. Each element of this filter is labeled with a letter that associates it with

the content of the sliding window with which its value is calculated.



This same effect can be observed in the convolutional neuronal network example that we are creating in this chapter. We start with an input image of 28×28 pixels and the resulting filters are 24×24 after the first convolution layer. And in the second convolution layer, we went from a 12×12 tensioner to an 8×8 tensioner.

But sometimes we want to obtain an output image of the same dimensions as the input and we can use the hyperparameter padding in the convolutional layers for this. With padding we can add zeros around the input images before sliding the window through it. In our case in the previous figure, for the output filter to have the same size as the input image, we can add a column to the right, a column to the left, a row above and a row below to the input image of zeros. Visually it can be seen in the following figure:

0	0	0	0	0	0	0
0	1	2	3	4	5	0
0	6	7	8	9	10	0
0	11	12	13	14	15	0
0	16	17	18	19	20	0
0	21	22	23	24	25	0
0	0	0	0	0	0	0

If we now slide the 3×3 window, we see that it can move 4 positions to the right and 4 positions down, generating the 25 windows that generate the filter size 5×5.

0	0	0
0	1	2
0	6	7

0	0	0
1	2	3
6	7	8

0	0	0
2	3	4
7	8	9

0	0	0
3	4	5
8	9	10

0	0	0
4	5	0
9	10	0

0	1	2
0	6	7
0	11	12

1	2	3
6	7	8
11	12	13

2	3	4
7	8	9
12	13	14

3	4	5
8	9	10
13	14	15

4	5	0
9	10	0
14	15	0

0	6	7
0	11	12
0	16	17

6	7	8
11	12	13
16	17	18

7	8	9
12	13	14
17	18	19

8	9	10
13	14	15
18	19	20

9	10	0
14	15	0
19	20	0

0	11	12
0	16	17
0	21	22

11	12	13
16	17	18
21	22	23

12	13	14
17	18	19
22	23	24

13	14	15
18	19	20
23	24	25

14	15	0
19	20	0
24	25	0

0	16	17
0	21	22
0	0	0

16	17	18
21	22	23
0	0	0

17	18	19
22	23	24
0	0	0

18	19	20
23	24	25
0	0	0

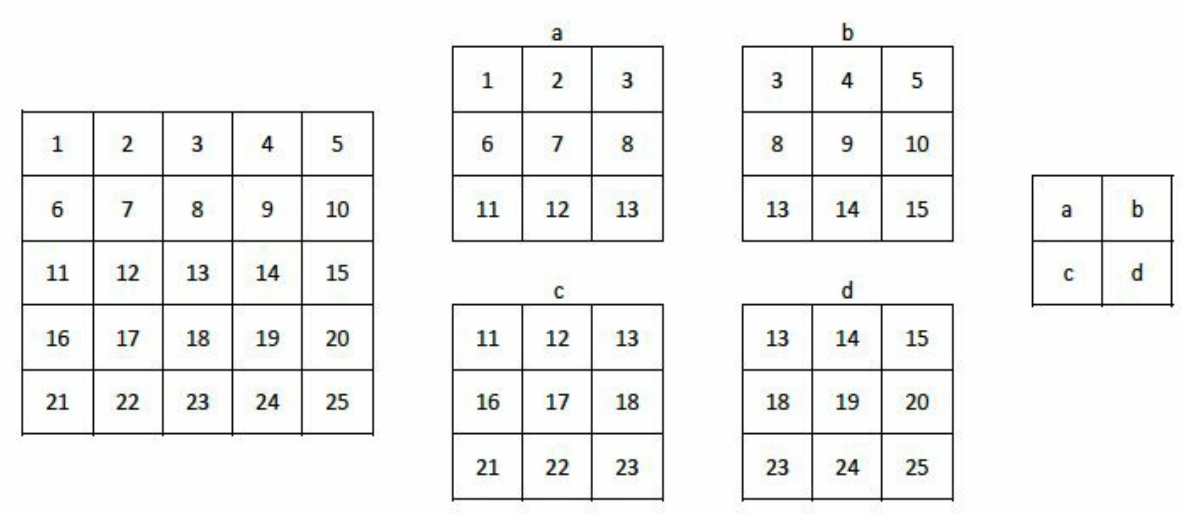
19	20	0
24	25	0
0	0	0

In Keras, the padding in the Conv2D layer is configured with the padding argument, which can have two values: "same", which indicates that as many rows and columns of zeros are added as necessary so that the output has the same dimension as the entry; and "valid", which indicates no padding (which is the default value of this argument in Keras).

Stride

Another hyperparameter that we can specify in a convolutional layer is the stride, which indicates the number of steps in which the filter window moves (in the previous example, the stride was one).

Large stride values decrease the size of the information that will be passed to the next layer. In the following figure we can see the same previous example but now with a stride value of 2:



As we can see, the 5×5 image has become a smaller 2×2 filter. But in reality

convolutional strides to reduce sizes are rarely used in practice; for this, the pooling operations that we have presented before are used. In Keras, the stride in the Conv2D layer is configured with the stride argument, which defaults to the *strides=(1,1)* value, which separately indicates the progress in the two dimensions.

Appendix: notebooks

Chapter 3

```
import keras
keras.__version__
```

```
'2.1.3'
```

Preloading the data in Keras

```
from keras.datasets import mnist

# we get the data for train and test
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
print(x_train.ndim)
```

```
3
```

```
print(x_train.shape)
```

```
(60000, 28, 28)
```

```
print(x_train.dtype)
```

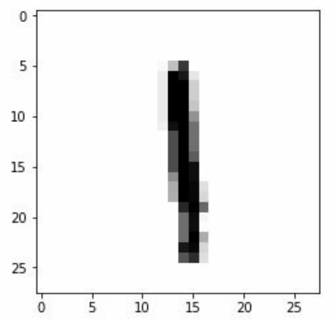
```
uint8
```

```
len (y_train)
```

60000

```
import matplotlib.pyplot as plt
plt.imshow(x_train[8], cmap=plt.cm.binary)
print(y_train[8])
```

1



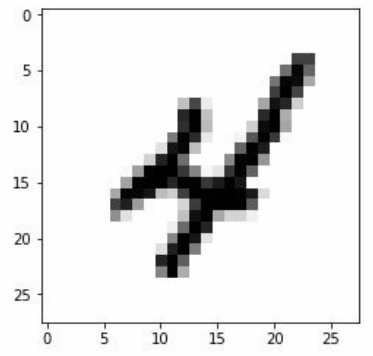

```
import numpy
from numpy import linalg
numpy.set_printoptions(precision=2, suppress=True, linewidth=120)
print(numpy.matrix(x_train[8]))
```

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   5  63 197   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 230  24   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254  48   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 255  48   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254  57   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254 108   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  16 239 254 143   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 143   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 143   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 162   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 240   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 113 254 240   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  83 254 245  31   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  79 254 246  38   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 214 254 150   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 241   8   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 240   2   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 254  82   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 230 247  40   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 168 209  31   0   0   0   0   0   0   0   0   0   0   0]
```

```
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

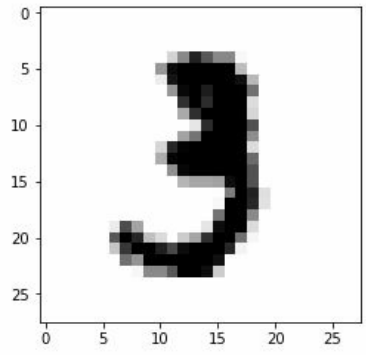
```
plt.imshow(x_train[9], cmap=plt.cm.binary)
print(y_train[9])
```

```
4
```



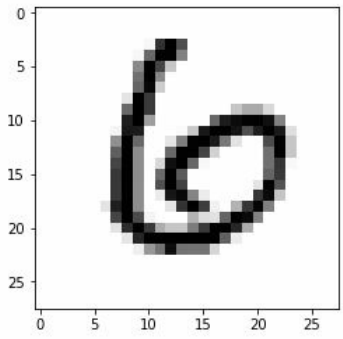
```
plt.imshow(x_train[10], cmap=plt.cm.binary)
print(y_train[10])
```

3



```
plt.imshow(x_test[11], cmap=plt.cm.binary)
print(y_test[11])
```

6


0
5
10
15
20
25
0
5
10
15
20
25

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

```
from keras.utils import to_categorical

y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

Model

```
from keras.models import Sequential
from keras.layers import Dense


model = Sequential()
model.add(Dense(10, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 10)                7850
_________________________________________________________________
dense_4 (Dense)              (None, 10)                110
```

```
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
_________________________________________________________________
```

Definition, training and evaluation

```
batch_size = 50
num_classes = 10
epochs=10

model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1
          )

test_loss, test_acc = model.evaluate(x_test, y_test)

print('Test loss:', test_loss)
print('Test accuracy:', test_acc)
```

```
Epoch 1/10
60000/60000 [==============================] - 2s 28us/step - loss: 2.0162 - acc: 0.4937
Epoch 2/10
60000/60000 [==============================] - 1s 24us/step - loss: 1.5334 - acc: 0.6890
Epoch 3/10
60000/60000 [==============================] - 1s 21us/step - loss: 1.1935 - acc: 0.7699
Epoch 4/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.9736 - acc: 0.8175
Epoch 5/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.8290 - acc: 0.8421
Epoch 6/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.7286 - acc: 0.8558
Epoch 7/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.6553 - acc: 0.8662
Epoch 8/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.5999 - acc: 0.8735
Epoch 9/10
60000/60000 [==============================] - 1s 20us/step - loss: 0.5569 - acc: 0.8782
Epoch 10/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.5229 - acc: 0.8822
10000/10000 [==============================] - 0s 18us/step
Test loss: 0.49781588258743287
Test accuracy: 0.8895
```

Predictions

```
predictions = model.predict(x_test)
```

```
import numpy as np
np.sum(predictions[11])
```

```
1.0
```

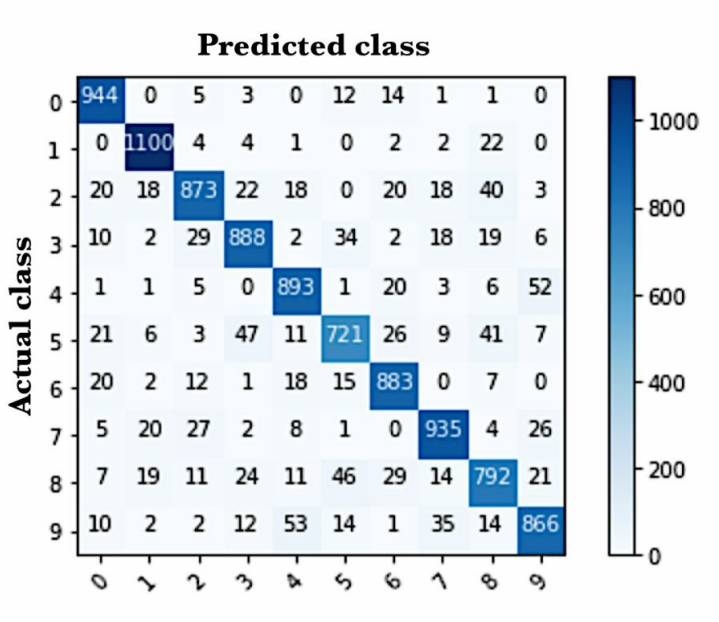
```
np.argmax(predictions[11])
```

```
6
```

```
#Note, this code is taken straight from the SKLEARN website

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('Observación')
    plt.xlabel('Predicción')
```

```
from collections import Counter
from sklearn.metrics import confusion_matrix
import itertools
```

```
# Predict the values from the validation dataset
Y_pred = model.predict(x_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test, axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(10))
```



Chapter 4

```
import keras
keras.__version__
```

'2.1.3'

```
from keras.datasets import mnist

(train_images, train_labels),(test_images, test_labels) = mnist.load_data()
# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
In [12]:
x_train = train_images.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255

from keras.utils import to_categorical

y_train = keras.utils.to_categorical(y_train, num_classes=10)
y_test = keras.utils.to_categorical(y_test, num_classes=10)
```

Base model

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import sgd

model = Sequential()
model.add(Dense(10, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 10)                7850
_________________________________________________________________
dense_12 (Dense)             (None, 10)                110
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
_________________________________________________________________
```

```
batch_size = 100
num_classes = 10
epochs=5

model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=0
          )

test_loss, test_acc = model.evaluate(x_test, y_test)

print('Test loss:', test_loss)
print('Test accuracy:', test_acc)
```

```
10000/10000 [==============================] - 0s 22us/step
Test loss: 1.2463368036270142
Test accuracy: 0.7618
```

ReLu activation function

```
batch_size = 100
num_classes = 10
epochs=5

model2 = Sequential()
model2.add(Dense(10, activation='relu', input_shape=(784,)))
model2.add(Dense(10, activation='softmax'))


model2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_17 (Dense)             (None, 10)                7850
_________________________________________________________________
dense_18 (Dense)             (None, 10)                110
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
_________________________________________________________________
```

```


model2.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model2.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=0
          )

test_loss, test_acc = model2.evaluate(x_test, y_test)

print('Model2 - Test loss:', test_loss)
print('Model2 - Test accuracy:', test_acc)
```

```
10000/10000 [==============================] - 0s 21us/step
Model2 - Test loss: 0.36076850221157075
Model2 - Test accuracy: 0.8998
```

512 nodes in Dense layer

```
model3 = Sequential()
model3.add(Dense(512, activation='relu', input_shape=(784,)))
model3.add(Dense(10, activation='softmax'))
model3.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_15 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_16 (Dense)             (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: c407,050
Non-trainable params: 0
_________________________________________________________________
```

```
model3.compile(loss='categorical_crossentropy',
               optimizer='sgd',
               metrics=['accuracy'])

epochs = 10
model3.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=0
           )

test_loss, test_acc = model3.evaluate(x_test, y_test)

print('Model3 - Test loss:', test_loss)
print('Model3 - Test accuracy:', test_acc)
```

```
10000/10000 [==============================] - 0s 40us/step
Model3 - Test loss: 0.24130829737782478
Model3 - Test accuracy: 0.9317
```

Chapter 6

```
import keras
keras.__version__
```

```
'2.1.3'
```

Basic elements of a convolutional neuronal network

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu',
        input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
=================================================================
Total params: 832
Trainable params: 832
Non-trainable params: 0
_________________________________________________________________
```

Basic CNN model

```
model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu',
        input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (5, 5), activation='relu'))
```

```
model.add(layers.MaxPooling2D((2, 2)))

model.summary()

```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 64)          51264
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 64)          0
=================================================================
Total params: 52,096
Trainable params: 52,096
Non-trainable params: 0
_________________________________________________________________
```

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

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 64)          51264
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                10250
=================================================================
Total params: 62,346
Trainable params: 62,346
Non-trainable params: 0
_________________________________________________________________
```

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels),(test_images, test_labels) = mnist.load_data()

print (train_images.shape)
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

(60000, 28, 28)

```
batch_size = 100
epochs = 5

model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(train_images, train_labels,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1
          )
```

```
Epoch 1/5
60000/60000 [==============================] - 55s 924us/step - loss: 0.1107 - acc: 0.9676
Epoch 2/5
60000/60000 [==============================] - 60s 1ms/step - loss: 0.0995 - acc: 0.9706
Epoch 3/5
60000/60000 [==============================] - 58s 965us/step - loss: 0.0916 - acc: 0.9729
```

```
Epoch 4/5
60000/60000 [==============================] - 54s 904us/step - loss: 0.0848 - acc: 0.9747
Epoch 5/5
60000/60000 [==============================] - 55s 914us/step - loss: 0.0801 - acc: 0.9762
```

Model evaluation

```
test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test loss:', test_loss)
print('Test accuracy:', test_acc)
```

```
10000/10000 [==============================] - 3s 277us/step
Test loss: 0.10979123749658465
Test accuracy: 0.9672
```

Acknowledgments

Writing a book requires motivation but also a lot of time and, for that, I want to start by saying thanks for the support and understanding that my family have shown with the fact that a laptop had to be shared many weekends and many nights spent with us.

To Ferran Julià, a great friend who has a degree in physics and also holds a degree in computer science from the UPC, I thank him for accompanying me in the writing of this book to improve its organization and reading.

In this line, I also want to thank Andrés Gómez of *La Fundación Pública Galega Centro Tecnolóxico de Supercomputación de Galicia* (CESGA) for his contribution to make a thorough review of the texts of this book.

To Juan Luís Domínguez, who came from Jerez to be able to read François Chollet's book together, I thank him for his help in preparing part of the codes that support this book. Without Juan Luis or Maurici Yagües, with whom we debated weekly on these subjects, this book would never have begun being written.

Special thanks to a colleague of the UPC and a great friend, Xavier Giró-i-Nieto. Xavier is an inexhaustible source of inspiration and knowledge, with whom I am co-directing the doctoral theses of Míriam Bellver and Victor Campos, to whom I also owe a great deal of thanks to for the knowledge that I have obtained on these topics and contents of this book. And I do not want to forget about other students like Xisco Sastre, whose final master's work also dealt with these concepts, being a great stimulus for my learning. A couple of research articles came from his results, of which I have used some graphs in this book.

There have been many experts on this subject that I do not personally know

who have also helped me at the time of writing, allowing me to share their ideas; therefore, I mention in detail the sources in the corresponding sections, more as a token of gratitude than for the reader having to consult this.

Among all the experts that inspired me, I must make special mention of François Chollet, Google researcher and creator of Keras, whom I have the good fortune to meet personally. This book is written after François published his book *Deep Learning with Python*, which has been of great help and inspiration.

My most sincere gratitude to all those who have read partially or totally the different drafts of this book: Bernat Torres (SOMmobilitat), Fernando García Sedano (Grupsa), Jordi Morera (UPCnet), Guifré Ballester (UPCnet), Sergi Sales (UPCnet) who have reported interesting comments to me to include in the version you have in your hands.

My greatest thanks to my university, the *Universitat Politècnica de Catalunya - UPC Barcelona Tech* and, especially, to Agustín Fernández Jiménez, vice-rector of Digital Transformation of the UPC for having written the foreword of this work and having made a detailed reading of the final draft. The UPC has been the working environment that has allowed me to carry out my research on these topics and accumulate the knowledge that I want to share here. A university that also offers me to give classes in the *Facultat d'Informàtica de Barcelona* (FIB) to some brilliant students, who encourage me to write books like this one.

As I mentioned at the beginning of the book, to researchers like Ricard Gavaldà or Jordi Nin, I owe it to them for awakening in me the interest in these subjects, years ago. As well as others, such as Rubèn Tous or Joan Capdevila, who accompanied me during the first steps in this topic.

I would like to thank very much the research center *Barcelona Supercomputing Center - National Computing Center* (BSC) and especially its directors Mateo Valero and Josep M. Martorell, and the directors of the

Computer Science department Jesús Labarta and Eduard Ayguadé, who have always allowed and supported me with this obsession that I have of needing to be “parant l'orella” to the technologies that will come.

In the promotion of this book, I thank the support of *FIB Alumni*, *Col·legi Oficial d'Enginyeria en Informàtica de Catalunya* (COEINF) and the technological information portal TECNONEWS. Also to Katy Wallace, for helping me naming this collection in which this work is edited.

My greatest thanks to our research group *Autonomic Systems and eBusiness Platforms* for the hard work they are doing in the projects that we carry out in the group: Adrià Correas, Carla Sauvanaud, Cesare Cugnasco, Eloy Gil, Enric Sosa, Fabrizio Pistagna, Joan Capdevila, Jordi Guitart, Juan Luís Dominguez, Maurici Yagues, Miriam Bellver, Paola Pardo, Pol Santamaria, Victor Campos and Yolanda Becerra.

My appreciation to my students of the 2018 academic year who have shared the first edition of this book in Spanish with me, the ones I call “cap problema” (or “no problem”, in English): Martín Acosta, Alessio Addimando, John Jairo Ballestas, Andrés Bermudez, Gabriel Cantos, Robert Carausu, Víctor Chamizo, Juan de los Reyes, Francesc, de Puig, Pau Figueras, Rafa Genés, Beñat Jimenez, Miquel Lara, Gil Laroussi, Tito Leiva, Irene Lopez, Eduardo Rodríguez, Isabel Samaniego, Marc Tula, Javier Vasquez, Marc Vila and Jonathan Zarama. Thank you all!

Finally, a sincere thank you to Nuria Rodríguez, Donnalee Shucan, Anna Llibre and Katy Wallace for the diligent English proofreading of this book.

About the author

Professor at the Universitat Politècnica de Catalunya Barcelona Tech[135] with 30 years of experience in teaching[136] and research[137] in high performance computing, with important scientific publications[138] and R&D projects[139] in companies and institutions.

His entrepreneurial spirit has led him to apply these systems in advanced analytics on Big Data, and at the moment his research focuses on supercomputing applied to Artificial Intelligence in general and Deep Learning in particular, an area in which he is co-advising three doctoral theses. His research has been published in the main forums of this field, including the *International Conference on Learning Representation* (ICLR 2018) and workshops within *Computer Vision and Pattern Recognition* (CVPR 2017, 2018) and *Neural Information Processing Systems* (NIPS 2016, 2017). In these publications he has collaborated with researchers from the UPC, Columbia University, Google and Facebook.

Since its inception he has led a research group[140] at the Barcelona Supercomputer Center[141]. He is intellectually eager and passionate about new technologies. During recent years he has carried out different activities contributing to the definition of what strategy to follow at a technological level ahead of the new challenges that these technologies represent. He is

currently a Board Member of iThinkUPC[142] & UPCnet, and acts as a trainer, mentor and expert[143] for various organizations and companies; In turn, he has also written several technical books[144], gives lectures[145] and has collaborated with different media[146], radio and television[147]. He has maintained his blog since 2006[148]. You can find more information at https://www.JordiTorres.org.

About Barcelona Supercomputing Center (BSC)

The Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS) is the leading supercomputing center in Spain. It houses MareNostrum, one of the most powerful supercomputers in Europe, and is a hosting member of the PRACE European distributed supercomputing infrastructure. The mission of BSC is to research, develop and manage information technologies in order to facilitate scientific progress. BSC combines HPC service provision and R&D into both computer and computational science (life, earth and engineering sciences) under one roof, and currently has over 500 staff from 44 countries. More information on page *www.bsc.es*

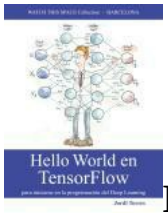
About Universitat Politècnica de Catalunya Barcelona Tech (UPC)

The Universitat Politècnica de Catalunya · BarcelonaTech (UPC) is a public institution dedicated to higher education and research, specialised in the fields of engineering, architecture and science. The activity that goes on at UPC campuses and schools has made the University a benchmark institution. Currently has 23,369 bachelor's students, 2,157 doctoral degree students, 5,338 master's degree students, 2,775 lifelong learning students and 5,068 staff members. The University harnesses the potential of basic and applied research, and transfers technology and knowledge to society. As a leading member of international networks of excellence, the UPC has a privileged relationship with global scientific and educational organisations. More information on page *www.upc.edu*

About Facultat d'Informàtica de Barcelona (FIB)

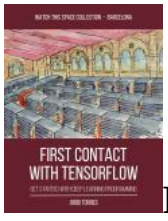
The *Facultat d'Informàtica de Barcelona* (Barcelona School of Informatics) is the reference center for computer studies since its inception in 1976, and the beginning of the educational activities during 1977-1978. Throughout these 40 years, the faculty has been in charge of the bachelor's degree, diploma courses, technical engineering and master degrees and currently, the formal qualifications in the field of Computer Science and related subjects. The *Facultat d'Informàtica de Barcelona*, in addition to bet on a solid scientific basis that will be useful throughout the professional career, promotes the professional skills essential in the labor market. That is why the faculty provides the students with teaching laboratories with the most modern equipment and classrooms with the latest technologies to support the most modern teaching methodologies. The agreements with companies and top research centers help to ensure almost full employment of the graduates. Therefore, the Bachelor and Master degrees of the *Facultat d'Informàtica de Barcelona* are accredited by the most prestigious international committees and the faculty has been recognized with several awards for quality and innovation in teaching. More information on page *www.fib.upc.edu*

WATCH THIS SPACE Book collection



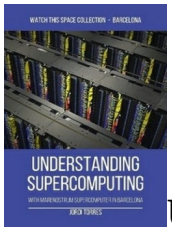
Hello World en TensorFlow
para iniciarse en la programación del Deep Learning

WATCH THIS SPACE collection – Barcelona : Book 1
Language: Spanish
ISBN 978-1-326-53238-3
January 2016
Library catalog: cataleg.upc.edu/record=b1472879
Freely available on-line: https://JordiTorres.org/TensorFlow



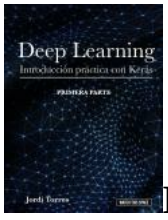
First Contact with TensorFlow
get started with deep learning programming

WATCH THIS SPACE collection – Barcelona: Book 2
Language: English
ISBN 978-1-326-56933-4
March 2016
Library catalog: cataleg.upc.edu/record=b1475713
Freely available on-line: https://JordiTorres.org/TensorFlow



Understanding Supercomputing
with Marenostrum supercomputer in Barcelona

WATCH THIS SPACE collection – Barcelona: Book 3
Language: English
ISBN 978-1-365-37682-5
September 2016
Library catalog: cataleg.upc.edu/record=b1490214
Freely available on-line: removed - deprecated (content: https://jorditorres.org/wp-content/uploads/2018/06/Understanding Supercomputing.content2016.pdf



Deep Learning
Introducción práctica con Keras

WATCH THIS SPACE collection – Barcelona: Book 4
Language: Spanish
ISBN 978-1-983-12981-0
June 2018
Library catalog: http://cataleg.upc.edu/record=b1508848
Freely available on-line: https://JordiTorres.org/DeepLearning/

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

[1] Jordi Torres, "Deep Learning - Introducción práctica con Keras". WATCH THIS SPACE collection – Barcelona. Book 4. Language: Spanish ISBN 978-1-983-12981-0 June 2018. Freely available online: http://JordiTorres.Barcelona/deeplearning/

[2] See more on Keras documentation pages available at:: https://keras.io

[3] François Chollet. Twitter account: https://twitter.com/fchollet (that you can see that he is very
active person on twitter).

[4] See https://jupyter.org

[5] See Github https://github.com/JordiTorresBCN/DEEP-LEARNING-practical-introduction-with-Keras

[6] See https://colab.research.google.com

[7] See https://research.google.com/colaboratory/faq.html

[8] Ricard Gavaldà web page [online]. Available at: http://www.lsi.upc.edu/~gavalda/

[9] Quoc Le and Marc'Aurelio Ranzato and Rajat Monga and Matthieu Devin and Kai Chen and Greg Corrado and Jeff Dean and Andrew Ng, *Building High-level Features Using Large Scale Unsupervised Learning* .International Conference in Machine Learning, ICML 2012 [online]. Available at: https://arxiv.org/abs/1112.6209 [Accessed: 12/02/2018]

[10] Blog de Jordi Torres. “Barcelona Supercomputing Center starts to work on Deep Learning” June 2014). [online]. Available at: http://jorditorres.org/barcelona-supercomputing-center-starts-to-works-on-deep-learning/

11] Moore’s Lay. Wikipedia. [online]. Available at: https://en.wikipedia.org/wiki/Moore%27s_law [Accessed: 12/03/2018]

[12] FLOPS. Wikipedia. [online]. Available at: https://en.wikipedia.org/wiki/FLOPS [Accessed: 12/03/2018]

[13] Top 500 List – November 2012. [online] Available at: https://www.top500.org/list/2012/11/?_ga=2.211333845.1311987907.1527961584-375587796.1527961584 [Accessed: 12/03/2018]

[14] Marenostrum 3. Barcelona Supercomputing Center. [online]. Available at: https://www.bsc.es/marenostrum/marenostrum/mn3 [Accessed: 12/03/2018]

[15] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada [online]. Available at:http://www.cs.toronto.edu/~kriz/imagenet_classification_with_deep_convolutional.pdf

[16] Russakovsky, O., Deng, J., Su, H. et al. Int J Comput Vis (2015) 115: 211. https://doi.org/10.1007/s11263-015-0816-y https://arxiv.org/abs/1409.0575

[17] Wikipedia. Graphics processing unit. [online] Available at: https://en.wikipedia.org/wiki/Graphics_processing_unit [Accessed: 12/02/2018]

[18] See https://www.top500.org/green500/

[19]Wikipedia. CUDA. [online]. Available at: https://en.wikipedia.org/wiki/CUDA

[20] See https://openai.com

[21] See https://blog.openai.com/ai-and-compute/

[22] See https://blog.openai.com/content/images/2018/05/compute_diagram-log@2x-3.png

[23] Campos, V., F. Sastre, M. Yagues, J. Torres, and X. Giro-I-Nieto. Scaling a Convolutional Neural Network for Classification of Adjective Noun Pairs with TensorFlow on GPU Clusters. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

[24] Campos, V., F. Sastre, M. Yagues, M. Bellver, X. Giro-I-Nieto, and J. Torres. Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster. Procedia Computer Science. Elsevier. Volume 108, Pag. 315-324. https://doi.org/10.1016/j.procs.2017.05.074

[25] MPI Wikipedia. https://en.wikipedia.org/wiki/Message_Passing_Interface

[26] Wikipedia. Tensor Processing Unit. [online]. Available at: https://en.wikipedia.org/wiki/Tensor_processing_unit [Accessed: 20/04/2018]

[27] Google IO 2018 conference (8-10 May 2018). Videos. [online]. Available at: https://www.youtube.com/playlist?list=PLOU2XLYxmsIInFRc3M44HUTQc3b_YJ4-Y [Accessed: 16/05/2018]

[28] See https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[29] Big Bets on A.I. Open a New Frontier for Chip Start-Ups, Too. The New York Times. January 14, 2018 [online]. Available at: https://www.nytimes.com/2018/01/14/technology/artificial-intelligence-chip-start-ups.html [Accessed: 20/04/2018]

[30] Marenostrum. Barcelona Supercomputing Center [online] Available at : https://www.bsc.es/marenostrum/marenostrum [Accessed: 20/05/2018]

[31] IBM Power System AC922 Introduction and Technocal Overview- IBM RedBooks by Andexandre Bicas Caldeira. March 2018. [online]. Available at http://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf [Accessed: 20/05/2018]

[32] IBM PowerAI: Deep Learning Unleashed on IBM Power Systems Servers. IBM RedBooks. March 2018. [online]. Available at https://www.dropbox.com/s/fd3oiuttqdeilut/IBMpowerAI.pdf?dl=0 [Accessed: 20/05/2018]

[33] Just at the time of finalizing the contents of this book NVIDIA has presented a new version of the V100 with 32 GB of memory. [online]. Available at https://www.top500.org/news/nvidia-refreshes-v100-gpus-upgrades-dgx-lineup/

[34] Tesla V100 NVIDIA. [online]. Available at http://www.nvidia.com/v100 [Accessed: 20/03/2018]

[35] CTE-POWER User's Guide. Barcelona Supercomputing Center 2018 [online]. Available at https://www.bsc.es/user-support/power.php [Accessed: 20/05/2018]

[36]

[37] See https://support.google.com/translate/answer/6142468?hl=en&ref_topic=7011659 [Accessed: 18/06/2018]. Use Google Chrome browser.

[38] Artificial Intelligence: A Modern Approach (AIMA) ·3rd edition, Stuart J Russell and Peter Norvig, Prentice hall, 2009. ISBN 0-13-604259-7

[39] Stuart J. Russell. Wikipedia. [online]. Available at: https://en.wikipedia.org/wiki/Stuart_J._Russell [Accessed: 16/04/2018]

[40] Peter Norvig Wikipedia. [online]. Available at: https://en.wikipedia.org/wiki/Peter_Norvig [Accessed: 16/04/2018]

[41] AI winter. Wikipedia. [online]. Available at: https://en.wikipedia.org/wiki/AI_winter [Accessed:

16/04/2018]

[42] Deep Learning. I Goodfellow, Y. Bengio and A Corville. MIT Press 2016. Also freely available online at http://www.deeplearningbook.org [consulta: 20/01/2018].

[43] The data that appear in this section are available at the time of writing this section (Spanish version of the book) at the beginning of the year 2018.

[44] Wikipedia, NoSQL. [online]. Available at: https://es.wikipedia.org/wiki/NoSQL [Accessed: 15/04/2018]

[45] The ImageNet Large Scale Visual Recognition Challenge (ILSVRC). [online]. Available at: www.image-net.org/challenges/LSVRC. [Accessed: 12/03/2018]

[46] MNIST [online]. Available at: http://yann.lecun.com/exdb/mnist/ [Accessed: 12/03/2018]

[47] STL [online]. Available at: http://ai.stanford.edu/~acoates/stl10/ [Accessed: 12/03/2018]

[48] See http://ccodataset.org

[49] See http://github.com/openimages/dataset

[50] See http://www.visualqa.org

[51] See http://ufldl.stanford.edu/housenumbers

[52] See http://www.cs.toronto.edu/~kriz/cifar.htmt

[53] See https://github.com/zalandoresearch/fashion-mnist

[54] See http://ai.stanford.edu/~amaas/data/sentiment

[55] See https://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups

[56] See https://archive.ics.uci.edu/ml/datasets/reuters-21578+text+categorization+collection

[57] See https://wordnet.princeton.edu

[58] See https://www.yelp.com/dataset

[59] See https://corpus.byu.edu/wiki

[60] See http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm

[61] See http://statmt.org/wmt11/translation-task.html

[62] See https://github.com/Jakobovski/free-spoken-digit-dataset

[63] See https://github.com/mdeff/fma

[64] See http://mtg.upf.edu/ismir2004/contest/tempoContest/node5.html

[65] See https://labrosa.ee.columbia.edu/millionsong

[66] See http://www.openslr.org/12

[67] See http://www.robots.ox.ac.uk/~vgg/data/voxceleb

[68] See https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

[69] See http://host.robots.ox.ac.uk/pascal/VOC/

[70] See https://www.plant-phenotyping.org/CVPPP2017

[71] See https://www.cityscapes-dataset.com

[72] Kaggle [online]. Available at: ttp://www.kaggle.com [Accessed: 12/03/2018]

[73] Empresas en la nube: ventajas y retos del Cloud Computing. Jordi Torres. Editorial Libros de Cabecera. 2011.

[74] Wikipedia. REST. [online]. Available at:

https://en.wikipedia.org/wiki/Representational_state_transfer [Accessed: 12/03/2018]

[75] Amazon ML [online]. Available at: https://aws.amazon.com/aml/ [Accessed: 12/03/2018]

[76] SageMaker [online]. Available at: https://aws.amazon.com/sagemaker/ [Accessed: 12/03/2018]

77] Azure ML Studio [online]. Available at: https://azure.microsoft.com/en-us/services/machine-learning-studio/ [Accessed: 12/03/2018]

[78] Azure Intelligent Gallery [online]. Available at: https://gallery.azure.ai [Accessed: 12/03/2018]

79] Google Prediction API [online]. Available at: https://cloud.google.com/prediction/docs/ [Accessed: 12/03/2018]

80] Google ML engine [online]. Available at: https://cloud.google.com/ml-engine/docs/technical-overview [Accessed: 12/03/2018]

[81] Watson Analytics [online]. Available at: https://www.ibm.com/watson/ [Accessed: 12/03/2018]

[82] PredicSis [online]. Available at: https://predicsis.ai [Accessed: 12/03/2018]

[83] BigML [online]. Available at: https://bigml.com [Accessed: 12/03/2018]

84] See https://www.kdnuggets.com/2018/02/top-20-python-ai-machine-learning-open-source-projects.html

[85] See https://github.com/tensorflow/tensorflow

[86] See https://keras.io

[87] See https://github.com/keras-team/keras

[88] See http://pytorch.org

[89] See http://www.openmp.org

[90] See https://github.com/pytorch/pytorch

[91] See http://scikit-learn.org

[92] See http://scikit-learn.org/stable/modules/preprocessing.html

[93] See https://github.com/scikit-learn/scikit-learn

[94] See http://deeplearning.net/software/theano

[95] See http://caffe.berkeleyvision.org

[96] See https://caffe2.ai

[97] See https://github.com/Microsoft/CNTK

[98] See https://mxnet.apache.org

[99] See https://deeplearning4j.org

[100] See https://chainer.org

[101] See https://developer.nvidia.com/digits

[102] See http://kaldi-asr.org/doc/dnn.html

[103] See https://lasagne.readthedocs.io/en/latest/

[104] See https://github.com/autumnai/leaf

[105] See http://www.vlfeat.org/matconvnet/

[106] See http://www.opendeep.org

[107] See https://github.com/dmlc/minerva

[108] See https://github.com/laonbud/SoooA/

[109] See https://arxiv.org

[110] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. Salakhutdinov "Improving neural networks by preventing co-adaptation of feature detectors" https://arxiv.org/pdf/1207.0580.pdf

[111] See https://twitter.com/ChrisFiloG/status/1009594246414790657

[112] The MNIST database of handwritten digits. [en línea]. Available at: http://yann.lecun.com/exdb/mnist [Consulta: 24/02/2017].

[113] Wikipedia, (2016). Antialiasing [en línea]. Available at: https://es.wikipedia.org/wiki/Antialiasing [Consulta: 9/01/2016].

[114] Wikipedia, (2018). Sigmoid function [en línea]. Available at: https://en.wikipedia.org/wiki/Sigmoid_function [Consulta: 2/03/2018].

[115] Wikipedia (2018). Perceptron [en línea]. Available at https://en.wikipedia.org/wiki/Perceptron [Consulta 22/12/2018]

[116] Wikipedia, (2018). Softmax function [en línea]. Available at: https://en.wikipedia.org/wiki/Softmax_function [Consulta: 22/02/2018].

[117] TensorFlow, (2016) Tutorial MNIST beginners. [en línea]. Available at: https://www.tensorflow.org/versions/r0.12/tutorials/mnist/beginners/ [Consulta: 16/02/2018].

[118] A color version of this figure can be found at www.JordiTorres.Barcelona/DeepLearning

[119] See https://keras.io/getting-started/functional-api-guide/

[120] See https://keras.io/models/sequential/

[121] https://keras.io/layers/core/#dense

[122] https://keras.io/initializers/#usage-of-initializers

[123] Confusion Matrix. Wikipedia. [online]. Available at: https://en.wikipedia.org/wiki/Confusion_matrix [Accessed: 30/04/2018]

[124] http://scikit-learn.org/stable/

[125] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

[126] See https://keras.io/activations/

[127] See https://en.wikipedia.org/wiki/Hyperbolic_function

[128] See https://keras.io/losses /

[129] See https://keras.io/optimizers/

[130] CS231n Convolutional Neural Networks for Visual Recognition. [online] Available at http://cs231n.github.io/neural-networks-2/#init [Accessed 20/04/2018]

[131] See http://keras.io/optimizers

[132] See http://playground.tensorflow.org

[133] Unsupervised learning of hierarchical representations with convolutional deep belief networks. H Lee, R. Grosse, R Ranganath and A. Y Ng.. Communications ACM, Vol 54 Issue 10, October 2011.

[134] GIMPS –8.2. Convolution Matrix [on-line] Available at: https://docs.gimp.org/en/plug-in-convmatrix.html [Accessed: 25/6/2018].

[135] See https://www.upc.edu/es?set_language=es

[136] See https://jorditorres.org/research-teaching/activity/
[137] See https://jorditorres.org/research-teaching/activity/
[138] See https://jorditorres.org/research-teaching/publications/
[139] See https://jorditorres.org/research-teaching/research-projects/
[140] See https://jorditorres.org/research-teaching/research-group/
[141] See https://www.bsc.es
[142] See https://www.ithinkupc.com/es
[143] See https://jorditorres.org/speaking-media/expert-advisor-consultant/
[144] See https://jorditorres.org/speaking-media/writer/
[145] See https://jorditorres.org/conferencias/
[146] See https://jorditorres.org/speaking-media/press-articles/
[147] See https://jorditorres.org/speaking-media/radio-tv/
[148] See https://jorditorres.org/blog/