ZACHARIAS VOULGARIS, PHD
YUNUS EMRAH BULUT

AI *for* DATA SCIENCE

ARTIFICIAL INTELLIGENCE FRAMEWORKS *and* FUNCTIONALITY *for* DEEP LEARNING, OPTIMIZATION, *and* BEYOND

TECHNICS PUBLICATIONS
TECHNOLOGY / LEADERSHIP

2 Lindsley Road
Basking Ridge, NJ 07920 USA
https://www.TechnicsPub.com

Cover design by Lorena Molinari

Edited by Sadie Hoberman and Lauren McCafferty

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the publisher, except for the inclusion of brief quotations in a review.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

All trade and product names are trademarks, registered trademarks, or service marks of their respective companies and are the property of their respective holders and should be treated as such.

First Edition

First Printing 2018

Copyright © 2018 Yunus Emrah Bulut and Zacharias Voulgaris, PhD

ISBN, print ed. 9781634624091
ISBN, PDF ed. 9781634624121

Library of Congress Control Number: 2018951809

Copyrighted material

Introduction

There is no doubt that *Artificial Intelligence* (commonly abbreviated *AI*) is making waves these days, perhaps more than the world anticipated as recently as the mid-2010s. Back then, AI was an esoteric topic that was too math-heavy to attract the average computer scientist, but now, it seems to be a household term. While it was once considered sci-fi lingo, it's now common to see and hear the term "AI" featured in ads about consumer products like smart phones.

This is to be expected, though; once an idea or technology reaches critical mass, it naturally becomes more acceptable to a wider audience, even if just on the application level. This level refers to what AI can do for us, by facilitating certain processes, or automating others. However, all this often gives rise to a series of misunderstandings. As AI itself has become more well-known, so have spread various ominous predictions about potential dangers of AI — predictions that are fueled by fear and fantasy, rather than fact.

Just like every other new technology, AI demands to be discussed with a sense of responsibility and ethical stature. An AI practitioner, especially one geared more towards the practical aspects of the field, understands the technology and its limitations, as well as the possible issues it has, which is why he talks about it without hyperbole and with projections of measured scope – that is, he projects realistic applications of AI, without talking about scenarios that resemble sci-fi films. After all, the main issues stemming for the misuse of a technology like this have more to do with the people using it, rather than the technology itself. If an AI system is programmed well, its risks are mitigated, and its outcomes are predictably positive.

About AI

But what exactly is AI? For starters, it's nothing like what sci-fi books

Copyrighted material

and films make it out to be. Modern AI technology helps to facilitate various processes in a more automatic and autonomous way, with little to no supervision from a human user. AI initiatives are realistic and purely functional. Although we can dream about what AI may evolve into someday, as AI practitioners, we focus on what we know and what we are certain about, rather than what could exist in a few decades.

AI comprises a set of *algorithms* that make use of information – mainly in the form of data – to make decisions and carry out tasks, much like a human would. Of course, the emulation of human intelligence is not an easy task; as such, the AIs of today are rudimentary and specialized. Despite their shortcomings, though, these modern systems can be particularly good at the tasks they undertake, even better than humans. For example, an *AI system*, which is a standalone program implementing one or more AI algorithms, that is created for identifying words from speech, can be more accurate than humans doing the same task.

It's important to note that all the AI systems we have today possess what is termed *narrow artificial intelligence*. This means that current AIs can do a limited set of tasks (or even just a single task) quite well, but offer at best mediocre performance at any other task. For instance, an AI might be great at figuring out your age based on a headshot, but that same AI almost certainly couldn't tell a classical music piece from a pop song.

Some AIs are designed to be used in robots, such as those designed for rescue missions, able to navigate various terrains. Other AIs are specialized in crunching data and facilitating various *data analytics* tasks. There are even AIs that emulate creative processes, like the AIs that generate artistic works, using the patterns they deduce from catalogs of existing work. *Chatbots* and other such AIs are focused solely on interacting with humans. The possibility of a more generalist AI (called *Artificial General Intelligence*, or *AGI*) exists, but it may take a while before it can manifest, or before we are ready to integrate it into our world.

Since all this may sound a bit abstract, let's clarify it a bit. If a system can make some decisions by capturing and analyzing signals related to the problem, that's an AI (sometimes termed "an AI system"). You've probably used an AI, even if you didn't know it. Online radios like

Copyrighted material

Spotify and Pandora use AI to recommend songs, and virtual assistants (like Siri) use AI to help you troubleshoot. Factors that help us decide whether a system is AI include the system's sophistication, its versatility, and how able it is to perform complex tasks.

Professor Alan Turing was the first to talk about this topic in a scientific manner. Upon studying this subject from both a theoretical and a practical perspective (through the creation of the first modern-day computer, used to crack the Enigma code in World War II[1]), he envisioned machines that could think and reason much like humans.

One of Professor Turing's most famous thought experiments is now named after him. The Turing test is a simple yet powerful *heuristic* for determining if a computer is advanced enough to manifest intelligence. This test involves taking either a human or a computer, and concealing it with another human. Another human, known as the examiner, then asks each of them a series of questions, without knowing which is which. If the examiner cannot determine from the answers to these questions whether he is speaking with a human or a computer, then the computer is said to have passed the test. This simple test has remained a standard for AI, still adding value to related research in the field in various ways.[2]

AI facilitates data science

So, how does AI fit within *data science?* After all, folks have been working in data science for years without these fancy AI algorithms. While it is certainly possible to gain valuable insights using traditional data science, AI-based algorithms can often bring about better performance in our *models* – the mathematical abstractions we create to simulate the phenomena we study. In highly competitive industries, this extra performance gained from AI-based *data models* can offer an edge over others in the market. Because many companies in these industries already have abundant data they can use to train the AI algorithms, we term them *AI-ready*.

AI is now far easier to apply than ever before. The early developers of AI have proven that AI can deliver a lot of value to the world, without

Copyrighted material

the help of a rocket scientist to make it work. This is largely thanks to a series of powerful AI *frameworks* and *libraries* that make AI methods more accessible to most data science practitioners.

In addition, AI has now diversified and matured enough to outperform conventional data science methods in many applications. For this, we must thank the increased computing resources at our disposal, particularly computing power. This is something made possible due to the *graphics processing units (GPUs)* becoming cheaper and easier to integrate to a computer, as add-ons. What's more, *cloud computing* has become more mainstream, enabling more people to have access to a virtual *computer cluster*, which they customize and rent, to run their AI projects. This makes AI systems easily scalable and cost-effective, while at the same time fostering experimentation and new use cases for this technology.

All this cooperation between AI and data science has led to a lot of research interest in AI. Research centers, individual researchers, and the R&D departments of various large companies have been investigating new ways to make these AI algorithms more scalable and more robust. This naturally boosts the field's impact on the world and makes it a more attractive technology—not just for the researchers, but for anyone willing to tinker with it, including many entrepreneurs in this field.

So yes, data science *could* continue to happen without AI. But in many cases, this wouldn't make much sense. It is now clear that the world of data science has a lot of problems and limitations that AI can help address. The overlap of these two closely related fields will only continue to grow as they both develop, so now is the perfect time to jump into learning AI with both feet.

About the book

This book covers various frameworks, focusing on the most promising ones, while also considering different AI algorithms that go beyond *deep learning*. Hopefully, this book will give you a more holistic understanding of the field of AI, arming you with a wide variety of tools (not just the ones currently in the limelight). With multiple tools at

Copyrighted material

your disposal, you can make your own decision about which one is best for any given data-related problem. After all, a good data scientist must not only know how to use each and every tool in the toolbox, but which tool is right for the job at hand.

Although most technologists and executives involved in data-driven processes can benefit significantly from this book, it is most suitable for data science professionals, AI practitioners, and those in related disciplines (such as *Python* or *Julia* programmers).

A basic understanding of data science is an important prerequisite to this book (for a thorough introduction to this, feel free to check out the book "Data Science Mindset, Methodologies, and Misconceptions" by Technics Publications). Moreover, a good mathematical background is recommended for those who want to dig deeper into the methods we examine in this book. Ultimately, though, the most important qualifications are determination and a curious nature, since you will ultimately put this knowledge into practice building your own AI systems.

Although this book is heavy on programming, you can still derive some useful understanding of AI, even if you don't do any coding. However, for best results, we recommend you work through the various examples and perhaps experiment a little on your own. We created a *Docker* image of all the code and data used in the book's examples, so you can follow along and experiment. See Appendix F for how to set up this environment and use the corresponding code.

This book provides an easy transition for someone with some understanding of the more well-known aspects of AI. As such, we start with an overview of the deep learning frameworks (chapter 1), followed by a brief description of the other AI frameworks, focusing on *optimization* algorithms and *fuzzy logic* systems (chapter 2). The objective of these first two chapters is to provide you with some frame of reference, before proceeding to the more hands-on and specialized aspects.

Namely, in chapter 3 we examine the MXNet framework for deep learning and how it works on Python. The focus here is on the most basic (and most widely used) deep learning systems, namely *feed forward neural networks* (also known as *multi-layer perceptrons*, or *MLPs* for short).

Copyrighted material

The two chapters that follow examine these deep learning systems using other popular frameworks: *Tensorflow* and *Keras*. All of the deep learning chapters contain some examples (provided as Jupyter notebooks with Python code in the Docker image) for hands-on practice on these systems.

Chapters 6 through 8 examine optimization algorithms, particularly the more advanced ones. Each chapter focuses on a particular framework of these algorithms, including *particle swarm optimization (PSO), genetic algorithms (GAs),* and *simulated annealing (SA).* We also consider applications of these algorithms, and how they can be of use in data science projects. The programming language we'll be using for these chapters is Julia (version 1.0), for performance reasons.[3]

After that, we look at more advanced AI methods as well as alternative AI systems. In chapter 9, specifically, we examine *convolutional neural networks (CNNs)* and *recurrent neural networks (RNNs)*, which are quite popular systems in the deep learning family. In chapter 10, we review *optimization ensembles*, which are not often discussed, but merit attention in this era of easy parallelization. Next, in chapter 11, we describe alternative AI frameworks for data science, such as *extreme learning machines (ELMs)* and *capsule networks (CapsNets)* which are either too new or too advanced for the mainstream of the AI field.

In the final chapter of the book, we mention about big data, data science specializations, and to help you practice we provide some sources of public datasets. The book concludes with some words of advice along with resources for additional learning. For example, in Appendix A we'll talk about *Transfer Learning*, while the topic of *Reinforcement Learning* will be covered in Appendix B. Autoencoder Systems will be briefly described in Appendix C, while *Generative Adversarial Networks (GANs)* will be introduced in Appendix D. Appendix E will take a look at the business aspect of AI in data science projects, while for those new to the Docker software, we have Appendix F.

This book contains a variety of technical terms, which are described in the glossary section that follows these chapters. Note that the first time a glossary term appears in the text of the book, it is marked in italics. The glossary also includes a few terms that are not mentioned in the text but are relevant.

Copyrighted material

The field of AI is vast. With this book, you can obtain a solid understanding of the field and hopefully some inspiration to explore it further as it evolves. So, let's get right to it, shall we?

https://bit.ly/2mEHUpc.

https://stanford.io/2useY7T.

For a brief tutorial on the language, you can watch this YouTube video: http://bit.ly/2Me0bsC.

Copyrighted material

CHAPTER 1

Deep Learning Frameworks

Deep learning is arguably the most popular aspect of AI, especially when it comes to data science (DS) applications. But what exactly are deep learning frameworks, and how are they related to other terms often used in AI and data science?

In this context, “framework” refers to a set of tools and processes for developing a certain system, testing it, and ultimately deploying it. Most AI systems today are created using frameworks. When a developer downloads and installs a framework on his computer, it is usually accompanied by a *library*. This library (or package, as it is often termed in high-level languages) will be compiled in the programming languages supported by the AI framework. The library acts like a proxy to the framework, making its various processes available through a series of functions and classes in the programming language used. This way, you can do everything the framework enables you to do, without leaving the programming environment where you have the rest of your scripts and data. So, for all practical purposes, that library is the framework, even if the framework can manifest in other programming languages too. This way, a framework supported by both Python and Julia can be accessed through either one of these languages, making the language you use a matter of preference. Since enabling a framework to function in a different language is a challenging task for the creators of the framework, oftentimes the options they provide for the languages compatible with that framework are rather limited.

But what is a *system*, exactly? In a nutshell, a system is a standalone program or script designed to accomplish a certain task or set of tasks. In a data science setting, a system often corresponds to a data model. However, systems can include features beyond just models, such as an I/O process or a data transformation process.

The term *model* involves a mathematical abstraction used to represent a real-

Copyrighted material

world situation in a simpler, more workable manner. Models in DS are optimized through a process called *training*, and validated through a process called *testing*, before they are deployed.

Another term that often appears alongside these terms is *methodology*, which refers to a set of methods and the theory behind those methods, for solving a particular type of problem in a certain field. Different methodologies are often geared towards different applications/objectives.

It's easy to see why frameworks are celebrities of sorts in the AI world. They help make the modeling aspect of the *pipeline* faster, and they make the *data engineering* demanded by deep learning models significantly easier. This makes AI frameworks great for companies that cannot afford a whole team of data scientists, or prefer to empower and develop the data scientists they already have.

These systems are fairly simple, but not quite "plug and play." In this chapter we'll explore the utility behind deep learning models, their key characteristics, how they are used, their main applications, and the methodologies they support.

About deep learning systems

Deep Learning (DL) is a subset of AI that is used for *predictive analytics*, using an AI system called an *Artificial Neural Network (ANN)*. Predictive analytics is a group of data science methodologies that are related to the prediction of certain variables. This includes various techniques such as classification, regression, etc. As for an ANN, it is a clever abstraction of the human brain, at a much smaller scale. ANNs manage to approximate every function (mapping) that has been tried on them, making them ideal for any data analytics related task. In data science, ANNs are categorized as machine learning methodologies.

The main drawback DL systems have is that they are "black boxes." It is exceedingly difficult – practically unfeasible – to figure out exactly how their predictions happen, as the data flux in them is extremely complicated.

Deep Learning generally involves large ANNs that are often specialized for specific tasks. Convolutional Neural Networks (CNNs) ANNs, for instance, are better for processing images, video, and audio data streams. However, all DL systems share a similar structure. This involves elementary modules

Copyrighted material

called *neurons* organized in layers, with various connections among them. These modules can perform some basic transformations (usually non-linear ones) as data passes through them. Since there is a plethora of potential connections among these neurons, organizing them in a structured way (much like real neurons are organized in network in brain tissue), we can obtain a more robust and function form of these modules. This is what an artificial neural network is, in a nutshell.

In general, DL frameworks include tools for building a DL system, methods for testing it, and various other Extract, Transform, and Load (ETL) processes; when taken together, these framework components help you seamlessly integrate DL systems with the rest of your pipeline. We'll look at this in more detail later in this chapter.

Although deep learning systems share some similarities with machine learning systems, certain characteristics make them sufficiently distinct. For example, conventional machine learning systems tend to be simpler and have fewer options for training. DL systems are noticeably more sophisticated; they each have a set of *training algorithms,* along with several parameters regarding the systems' *architecture*. This is one of the reasons we consider them a distinct framework in data science.

DL systems also tend to be more autonomous than their machine counterparts. To some extent, DL systems can do their own *feature engineering*. More conventional systems tend to require more fine-tuning of the feature-set, and sometimes require *dimensionality reduction* to provide any decent results.

In addition, the *generalization* of conventional ML systems when provided with additional data generally don't improve as much as DL systems. This is also one of the key characteristics that makes DL systems a preferable option when *big data* is involved.

Finally, DL systems take longer to train and require more computational resources than conventional ML systems. This is due to their more sophisticated functionality. However, as the work of DL systems is easily *parallelizable*, modern computing architecture as well as cloud computing, benefit DL systems the most, compared to other predictive analytics systems.

How deep learning systems work

Copyrighted material

At their cores, all DL frameworks work similarly, particularly when it comes to the development of DL networks. First, a DL network consists of several neurons organized in layers; many of these are connected to other neurons in other layers. In the simplest DL network, connections take place only between neurons in adjacent layers.

The first layer of the network corresponds to the features of our dataset; the last layer corresponds to its outputs. In the case of *classification*, each class has its own node, with node values reflecting how confident the system is that a data point belongs to that class. The layers in the middle involve some combination of these features. Since they aren't visible to the end user of the network, they are described as *hidden* (see Figure 1).

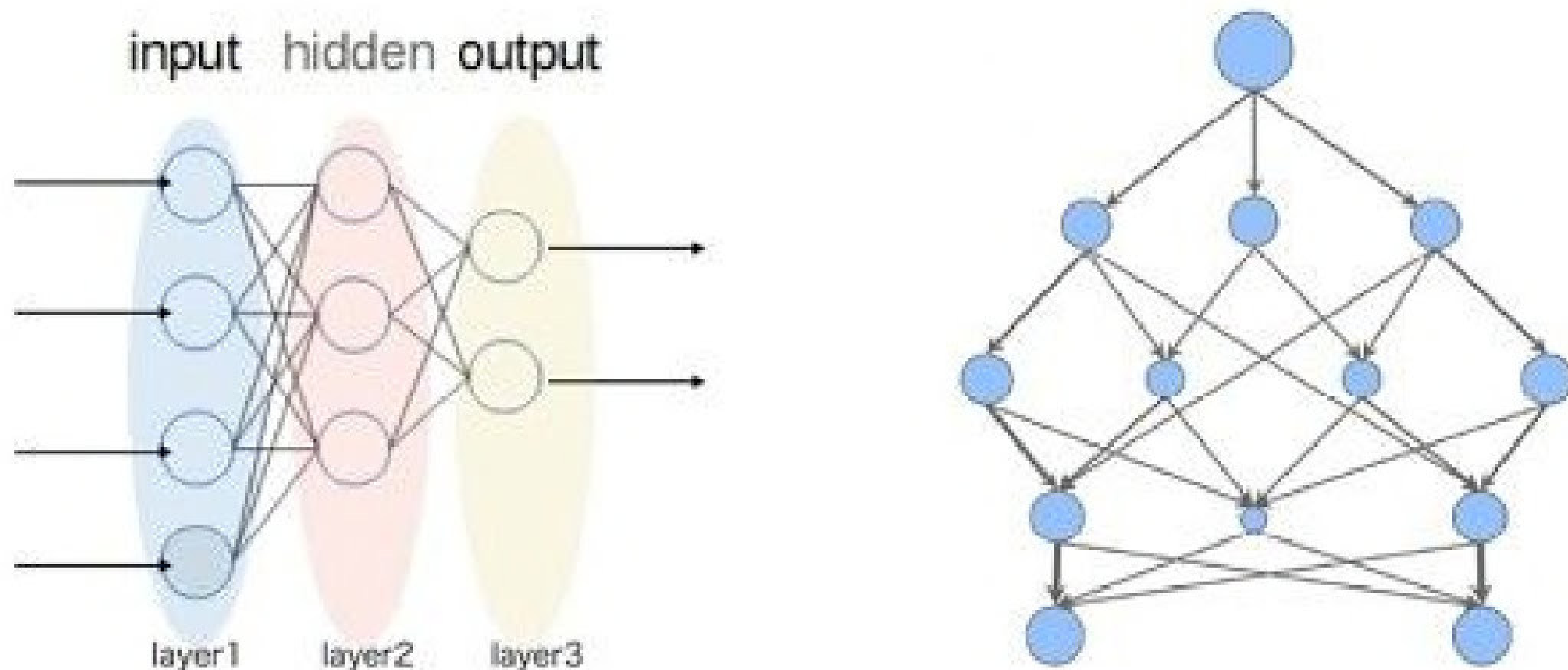

Figure 1. Depictions of a couple of simplified deep learning networks. There are often more hidden layers than seen in these examples. The number of neurons in the output layer can be arbitrary, depending on the *target variable*. Images created by *Richmanokada* and *AlexNet22*, respectively, and made available through CC licensing. The image on the right has been processed for additional comprehensiveness.

The connections among the nodes are weighted, indicating the contribution of each node to the nodes of the next layer it is connected to, in the next layer. The weights are initially randomized, when the network object is created, but are refined as the ANN is trained.

Moreover, each node contains a mathematical function that creates a

Copyrighted material

transformation of the received signal, before it is passed to the next layer. This is referred to as the *transfer function* (also known as the *activation function*). The *sigmoid function* is the most well-known of these, but others include *softmax*, *tanh*, and *ReLU*. We'll delve more into these in a moment.

Furthermore, each layer has a *bias node*, which is a constant that appears unchanged on each layer. Just like all the other nodes, the bias node has a weight attached to its output. However, it has no transfer function. Its weighted value is simply added to the other nodes it is connected to, much like a constant c is added to a *regression* model in Statistics. The presence of such a term balances out any bias the other terms inevitably bring to the model, ensuring that the overall bias in the model is minimal. As the topic of bias is a very complex one, we recommend you check out some external resources[4] if you are not familiar with it.

Once the transformed inputs (features) and the biases arrive at the end of the DL network, they are compared with the *target variable*. The differences that inevitably occur are relayed back to the various nodes of the network, and the weights are changed accordingly. Then the whole process is repeated until the error margin of the outputs is within a certain predefined level, or until the maximum number of iterations is reached. Iterations of this process are often referred to as training *epochs*, and the whole process is intimately connected to the training algorithm used. In fact, the number of epochs used for training a DL network is often set as a parameter and it plays an important role in the ANN's performance.

All of the data entering a neuron (via connections with neurons of the previous layer, as well as the bias node) is summed, and then the transfer function is applied to the sum, so that the data flow from that node is $y = f(\Sigma(w_i x_i + b))$, where w_i is the weight of node i of the previous layer, and x_i its output, while b is the bias of that layer. Also, f() is the mathematical expression of the transfer function.

This relatively simple process is at the core of every ANN. The process is equivalent to that which takes place in a *perceptron* system—a rudimentary AI model that emulates the function of a single neuron. Although a perceptron system is never used in practice, it is the most basic element of an ANN, and the first system created using this paradigm.

The function of a single neuron is basically a single, predefined transformation of the data at hand. This can be viewed as a kind of meta-feature of the framework, as it takes a certain input x and after applying a (usually non-linear) function f() to it, x is transformed into something else,

Copyrighted material

which is the neuron's output y.

While in the majority of cases one single meta-feature would be terrible at predicting the target variable, several of them across several layers can work together quite effectively – no matter how complex the mapping of the original features to the target variable. The downside is that such a system can easily *overfit*, which is why the training of an ANN doesn't end until the error is minimal (smaller than a predefined threshold).

This most rudimentary description of a DL network works for networks of the multi-layer perceptron type. Of course, there are several variants beyond this type. CNNs, for example, contain specialized layers with huge numbers of neurons, while RNNs have connections that go back to previous layers. Additionally, some training algorithms involve *pruning* nodes of the network to ensure that no overfitting takes place.

Once the DL network is trained, it can be used to make predictions about any data similar to the data it was trained on. Furthermore, its generalization capability is quite good, particularly if the data it is trained on is diverse. What's more, most DL networks are quite robust when it comes to noisy data, which sometimes helps them achieve even better generalization.

When it comes to classification problems, the performance of a DL system is improved by the class boundaries it creates. Although many conventional ML systems create straightforward boundary landscapes (e.g. rectangles or simple curves), a DL system creates a more sophisticated line around each class (reminiscent of the borders of certain counties in the US). This is because the DL system is trying to capture every bit of signal it is given in order to make fewer mistakes when classifying, boosting its raw performance. Of course, this highly complex mapping of the classes makes interpretation of the results a very challenging, if not unfeasible, task. More on that later in this chapter.

Main deep learning frameworks

Having knowledge of multiple DL frameworks gives you a better understanding of the AI field. You will not be limited by the capabilities of a specific framework. For example, some DL frameworks are geared towards a certain programming language, which may make focusing on just that framework an issue, since languages come and go. After all, things change very rapidly in technology, especially when it comes to software. What better

Copyrighted material

way to shield yourself from any unpleasant developments than to be equipped with a diverse portfolio of DL know-how?

The main frameworks in DL include MXNet, TensorFlow, and Keras. Pytorch and Theano have also played an important role, but currently they are not as powerful or versatile as the aforementioned frameworks, which we will focus on in this book. Also, for those keen on the Julia language, there is the *Knet* framework, which to the best of our knowledge, is the only deep learning framework written in a high-level language mainly (in this case, Julia). You can learn more about it at its Github repository.[5]

MXNet is developed by Apache and it's Amazon's favorite framework. Some of Amazon's researchers have collaborated with researchers from the University of Washington to benchmark it and make it more widely known to the scientific community. We'll examine this framework in Chapter 3.

TensorFlow is probably the most well-known DL framework, partly because it has been developed by Google. As such, it is widely used in the industry and there are many courses and books discussing it. In Chapter 4 we'll delve into it more.

Keras is a high-level framework; it works on top of TensorFlow (as well as other frameworks like Theano). Its ease of use without losing flexibility or power makes it one of the favorite deep learning libraries today. Any data science enthusiast who wants to dig into the realm of deep learning can start using Keras with reasonably little effort. Moreover, Keras' seamless integrity with TensorFlow, plus the official support it gets from Google, have convinced many that Keras will be one of the long-lasting frameworks for deep learning models, while its corresponding library will continue to be maintained. We'll investigate it in detail in Chapter 5.

Main deep learning programming languages

As a set of techniques, DL is language-agnostic; any computer language can potentially be used to apply its methods and construct its data structures (the DL networks), even if each DL framework focuses on specific languages only. This is because it is more practical to develop frameworks that are compatible with certain languages, some programming languages are used more than others, such as Python. The fact that certain languages are more commonly used in data science plays an important role in language selection, too. Besides, DL is more of a data science framework nowadays anyway, so it is

Copyrighted material

marketed to the data science community mainly, as part of Machine Learning (ML). This likely contributes to the confusion about what constitutes ML and AI these days.

Because of this, the language that dominates the DL domain is Python. This is also the reason why we use it in the DL part of this book. It is also one of the easiest languages to learn, even if you haven't done any programming before. However, if you are using a different language in your everyday work, there are DL frameworks that support other languages, such as Julia, *Scala*, R, JavaScript, Matlab, and Java. Julia is particularly useful for this sort of task as it is high-level (like Python, R, and Matlab), but also very fast (like any low-level language, including Java).

In addition, almost all the DL frameworks support C / C++, since they are usually written in C or its object-oriented counterpart. Note that all these languages access the DL frameworks through *APIs*, which take the form of packages in these languages. Therefore, in order to use a DL framework in your favorite language's environment, you must become familiar with the corresponding package, its classes, and its various functions. We'll guide you through all that in chapters 3 to 5 of this book.

How to leverage deep learning frameworks

Deep learning frameworks add value to AI and DS practitioners in various ways. The most important value-adding processes include *ETL* processes, building data models, and deploying these models. Beyond these main functions, a DL framework may offer other things that a data scientist can leverage to make their work easier. For example, a framework may include some visualization functionality, helping you produce some slick graphics to use in your report or presentation. As such, it's best to read up on each framework's documentation, becoming familiar with its capabilities to leverage it for your data science projects.

ETL processes

A DL framework can be helpful in fetching data from various sources, such as databases and files. This is a rather time-consuming process if done manually, so using a framework is very advantageous. The framework will also do

Copyrighted material

some formatting on the data, so that you can start using it in your model without too much data engineering. However, doing some data processing of your own is always useful, particularly if you have some domain knowledge.

Building data models

The main function of a DL framework is to enable you to efficiently build data models. The framework facilitates the architecture design part, as well as all the data flow aspects of the ANN, including the training algorithm. In addition, the framework allows you to view the performance of the system as it is being trained, so that you gain insight about how likely it is to overfit.

Moreover, the DL framework takes care of all the testing required before the model is tested on different than the dataset it was trained on (new data). All this makes building and fine-tuning a DL data model a straightforward and intuitive process, empowering you to make a more informed choice about what model to use for your data science project.

Deploying data models

Model deployment is something that DL frameworks can handle, too, making movement through the data science pipeline swifter. This mitigates the risk of errors through this critical process, while also facilitating easy updating of the deployed model. All this enables the data scientist to focus more on the tasks that require more specialized or manual attention. For example, if you (rather than the DL model) worked on the feature engineering, you would have a greater awareness of exactly what is going into the model.

Deep learning methodologies and applications

Deep learning is a very broad AI category, encompassing several data science methodologies through its various systems. As we have seen, for example, it can be successfully used in classification—if the output layer of the network is built with the same number of neurons as the number of classes in the dataset. When DL is applied to problems with the regression methodology, things are simpler, as a single neuron in the output layer is enough. *Reinforcement learning* is another methodology where DL is used; along with

Copyrighted material

the other two methodologies, it forms the set of *supervised learning,* a broad methodology under the predictive analytics umbrella (see Appendix B).

DL is also used for dimensionality reduction, which (in this case) comprises a set of meta-features that are usually developed by an *autoencoder* system (see Appendix C for more details on this kind of DL network). This approach to dimensionality reduction is also more efficient than the traditional statistical ones, which are computationally expensive when the number of features is remarkably high. *Clustering* is another methodology where deep learning can be used, with the proper changes in the ANN's structure and data flow. Clustering and dimensionality reduction are the most popular *unsupervised learning* methodologies in data science and provide a lot of value when exploring a dataset. Beyond these data science methodologies involving DL, there are others that are more specialized and require some domain expertise. We'll talk about some of them more, shortly.

There are many applications of deep learning. Some are more established or general, while others are more specialized or novel. Since DL is still a new tool, its applications in the data science world remain works in progress, so keep an open mind about this matter. After all, the purpose of all AI systems is to be as universally applicable as possible, so the list of applications is only going to grow.

For the time being, DL is used in complex problems where high-accuracy predictions are required. These could be datasets with high dimensionality and/or highly non-linear patterns. In the case of high-dimensional datasets that need to be summarized into a more compact form with fewer dimensions, DL is a highly effective tool for the job. Also, since the very beginning of its creation, DL has been applied to image, sound, and video analytics, with a focus on images. Such data is quite difficult to process otherwise; the tools used before DL could only help so much, and developing those features manually was a very time-consuming process.

Moving on to more niche applications, DL is widely used in various *natural language processing (NLP)* methods. This includes all kinds of data related to everyday text, such as that found in articles, books, and even social media posts. Where it is important to identify any positive or negative attitudes in the text, we use a methodology called "*sentiment analysis*," which offers a fertile ground for many DL systems. There are also DL networks that perform text prediction, which is common in many mobile devices and some text editors. More advanced DL systems manage to link images to captions by mapping these images to words that are relevant and that form sentences.

Copyrighted material

Such advanced applications of DL include chatbots, in which the AI system both creates text and understands the text it is given. Also, applications like text summarization are under the NLP umbrella too and DL contributes to them significantly. Some DL applications are more advanced or domain-specific – so much so that they require a tremendous amount of data and computing power to work. However, as computing becomes more readily available, these are bound to become more accessible in the short term.

Assessing a deep learning framework

DL frameworks make it easy and efficient to employ DL in a data science project. Of course, part of the challenge is deciding which framework to use. Because not all DL frameworks are built equal, there are factors to keep in mind when comparing or evaluating these frameworks.

The number of languages supported by a framework is especially important. Since programming languages are particularly fluid in the data science world, it is best to have your language bases covered in the DL framework you plan to use. What's more, having multiple languages support in a DL framework enables the formation of a more diverse data science team, with each member having different specific programming expertise.

You must also consider the raw performance of the DL systems developed by the framework in question. Although most of these systems use the same low-level language on the back end, not all of them are fast. There may also be other overhead costs involved. As such, it's best to do your due diligence before investing your time in a DL framework—particularly if your decision affects other people in your organization.

Furthermore, consider the ETL processes supporting a DL framework. Not all frameworks are good at ETL, which is both inevitable and time-consuming in a data science pipeline. Again, any inefficiencies of a DL framework in this aspect are not going to be advertised; you must do some research to uncover them yourself.

Finally, the user community and documentation around a DL framework are important things, too. Naturally, the documentation of the framework is going to be helpful, though in some cases it may leave much to be desired. If there is a healthy community of users for the DL framework you are considering, things are bound to be easier when learning its more esoteric aspects—as well as when you need to troubleshoot issues that may arise.

Copyrighted material

Interpretability

Interpretability is the capability of a model to be understood in terms of its functionality and its results. Although interpretability is often a given with conventional data science systems, it is a pain point of every DL system. This is because every DL model is a "black box," offering little to no explanation for why it yields the results it does. Unlike the framework itself, whose various modules and their functionality is clear, the models developed by these frameworks are convoluted graphs. There is no comprehensive explanation as to how the inputs you feed them turn into the outputs they yield.

Although obtaining an accurate result through such a method may be enticing, it is quite hard to defend, especially when the results are controversial or carry a demographic bias. The reason for a demographic bias has to do with the data, by the way, so no number of bias nodes in the DL networks can fix that, since a DL network's predictions can only be as good as the data used to train it. Also, the fact that we have no idea how the predictions correspond to the inputs allows biased predictions to slip through unnoticed.

However, this lack of interpretability may be resolved in the future. This may require a new approach to them, but if it's one thing that the progress of AI system has demonstrated over the years, it is that innovations are still possible and that new architectures of models are still being discovered. Perhaps one of the newer DL systems will have interpretability as one of its key characteristics.

Model maintenance

Maintenance is essential to every data science model. This entails updating or even upgrading a model in production, as new data becomes available. Alternatively, the assumptions of the problem may change; when this happens, model maintenance is also needed. In a DL setting, model maintenance usually involves retraining the DL network. If the retrained model doesn't perform well enough, more significant changes may be considered such as changing the architecture or the training parameters. Whatever the case, this whole process is largely straightforward and not too time-consuming.

Copyrighted material

How often model maintenance is required depends on the dataset and the problem in general. Whatever the case, it is good to keep the previous model available too when doing major changes, in case the new model has unforeseen issues. Also, the whole model maintenance process can be automated to some extent, at least the offline part, when the model is retrained as new data is integrated with the original dataset.

When to use DL over conventional data science systems

Deciding when to use a DL system instead of a conventional method is an important task. It is easy to be enticed by the new and exciting features of DL, and to use it for all kinds of data science problems. However, not all problems require DL. Sometimes, the extra performance of DL is not worth the extra resources required. In cases where conventional data science systems fail, or don't offer any advantage (like interpretability), DL systems may be preferable. Complex problems with lots of variables and cases with non-linear relationships between the features and the target variables are great matches for a DL framework.

If there is an abundance of data, and the main objective is good raw performance in the model, a DL system is typically preferable. This is particularly true if computational resources are not a concern, since a DL system requires quite a lot of them, especially during its training phase. Whatever the case, it's good to consider alternatives before setting off to build a DL model. While these models are incredibly versatile and powerful, sometimes simpler systems are good enough.

Summary

- Deep Learning is a particularly important aspect of AI, and has found a lot of applications in data science.
- Deep Learning employs a certain kind of AI system called an Artificial Neural Networks (or ANN). An ANN is a graph-based system involving a series of (usually non-linear) operations, whereby the original features are transformed into a few meta-features capable of predicting the target variable more accurately than the original features.

Copyrighted material

- The main frameworks in DL are MXNet, TensorFlow, and Keras, though Pytorch and Theano also play roles in the whole DL ecosystem. Also, Knet is an interesting alternative for those using Julia primarily.
- There are various programming languages used in DL, including Python, Julia, Scala, Javascript, R, and C / C++. Python is the most popular.
- A DL framework offers diverse functionality, including ETL processes, building data models, deploying and evaluating models, and other functions like creating visuals.
- A DL system can be used in various data science methodologies, including Classification, Regression, Reinforcement Learning, Dimensionality Reduction, Clustering, and Sentiment Analysis.
- Classification, regression, and reinforcement learning are supervised learning methodologies, while dimensionality reduction and clustering are unsupervised.
- Applications of DL include making high-accuracy predictions for complex problems; summarizing data into a more compact form; analyzing images, sound, or video; natural language processing and sentiment analysis; text prediction; linking images to captions; chatbots; and text summarization.
- A DL framework needs to be assessed on various metrics (not just popularity). Such factors include the programming languages it supports, its raw performance, how well it handles ETL processes, the strength of its documentation and user communities, and the need for future maintenance.
- It is not currently very easy to interpret DL results and trace them back to specific features (i.e. DL results currently have low *interpretability*).
- Giving more weight to raw performance or interpretability can help you decide whether a DL system or conventional data science system is ideal for your particular problem. Other factors, like the amount of computational resources at our disposal, are also essential for making this decision.

A good starting point can be found at http://bit.ly/2vzKC30.

Copyrighted material

https://github.com/denizyuret/Knet.jl.

Copyrighted material

CHAPTER 2

AI Methodologies Beyond Deep Learning

As we've seen, deep learning is a key aspect of most robust AI systems—but it's not the only way to use AI. This chapter covers some alternatives to deep learning. Even if these methods are not as popular as DL methods, they can be very useful in certain scenarios. We'll take a look at the two main methodologies – optimization and fuzzy logic – as well as some less well-known methods such as *artificial creativity*. We'll cover new trends in AI methodologies. Finally, we'll explore some useful considerations to leverage these methods and make the most out of them for your data science projects.

Many of the AI methodologies alternative to DL don't use ANNs of any kind, but rely on other systems that exhibit a certain level of intelligence. As some such systems don't use an obscure graph for making their predictions (like ANNs do) they are more transparent than DL, making them useful when interpreting results. Most of these alternative AI methodologies have been around for a few decades now, so there is plenty of support behind them, making them reliable resources overall. Others are generally newer, but are quite robust and reliable nevertheless.

Since the field of AI is rapidly evolving, these alternatives to DL may become even more relevant over the next few years. After all, many data science problems involve optimizing a function.

Among the various alternative AI methodologies out there, the ones that are more suitable for data science work can be classified under the *optimization* umbrella. However, fuzzy logic systems may be useful,

Copyrighted material

even though they apply mainly to low-dimensionality datasets, as we'll see later. Optimization, on the other hand, involves all kinds of datasets, and is often used within other data science systems.

Optimization

Optimization is the process of finding the maximum or minimum of a given function (also known as a *fitness function)*, by calculating the best values for its variables (also known as a "solution"). Despite the simplicity of this definition, it is not an easy process; often involves restrictions, as well as complex relationships among the various variables. Even though some functions can be optimized using some mathematical process, most functions we encounter in data science are not as simple, requiring a more advanced technique.

Optimization systems (or *optimizers,* as they are often referred to) aim to optimize in a systematic way, oftentimes using a heuristics-based approach. Such an approach enables the AI system to use a macro level concept as part of its low-level calculations, accelerating the whole process and making it more light-weight. After all, most of these systems are designed with scalability in mind, so the heuristic approach is most practical.

Importance of optimization

Optimization is especially important in many data science problems—particularly those involving a lot of variables that need to be fine-tuned, or cases where the conventional tools don't seem to work. In order to tackle more complex problems, beyond classical methodologies, optimization is essential. Moreover, optimization is useful for various data engineering tasks such as *feature selection*, in cases where maintaining a high degree of interpretability is desired. We'll investigate the main applications of optimizers in data science later in this chapter.

Copyrighted material

Optimization systems overview

There are different kinds of optimization systems. The most basic ones have been around the longest. These are called "deterministic optimizers," and they tend to yield the best possible solution for the problem at hand. That is, the absolute maximum or minimum of the fitness function. Since they are quite time-consuming and cannot handle large-scale problems, these deterministic optimizers are usually used for applications where the number of variables is relatively small. A classic example of such an optimizer is the one used for least squared error regression—a simple method to figure out the optimal line that fits a set of data points, in a space with relatively small dimensionality.

In addition to deterministic optimizers, there are *stochastic* optimizers, which more closely fit the definition of AI. After all, most of these are based on natural phenomena, such as the movement of the members of a *swarm*, or the way a metal melts. The main advantage of these methods is that they are very efficient. Although they usually don't yield the absolute maximum or minimum of the function they are trying to optimize, their solutions are good enough for all practical purposes (even if they vary slightly every time you run the optimizer). Stochastic optimizers also scale very well, so they are ideal for complex problems involving many variables. In this book we will focus on some of these stochastic optimization methods.

Programming languages for optimization

Optimization is supported by most programming languages in terms of libraries, like the *Optim* and *JuMP* packages in Julia. However, each algorithm is simple enough so that you can code it yourself, if you cannot find an available "off-the-shelf" function. In this book we'll examine the main algorithms for advanced optimization and how they are implemented in Julia. We chose this programming language because it combines ease of use and high execution speed. Remember that all the code is available in the Docker environment that accompanies this book.

Copyrighted material

Fuzzy inference systems

Fuzzy logic (FL) is a methodology designed to emulate the human capacity of imprecise or approximate reasoning. This ability to judge under uncertainty was previously considered strictly human, but FL has made it possible for machines, too.

Despite its name, there is nothing unclear about the outputs of fuzzy logic. This is because fuzzy logic is an extension of classical logic, when partial truths are included to extend bivalued logic (true or false) to a multivalued logic (degrees of truth between true and false).

According to its creator, Professor Zadeh, the ultimate goal of fuzzy logic is to form the theoretical foundation for reasoning about imprecise propositions (also known as "approximate reasoning"). Over the past couple decades, FL has gained ground and become regarded as one of the most promising AI methodologies.

A FL system contains a series of mappings corresponding to the various features of the data at hand. This system contains terms that make sense to us, such as high-low, hot-cold, and large-medium-small, terms that may appear fuzzy since there are no clear-cut boundaries among them. Also, these attributes are generally relative and require some context to become explicit, through a given mapping between each term and some number that the system can use in its processes. This mapping is described mathematically through a set of membership functions, graphically taking the form of triangles, trapezoids, or even curves. This way something somewhat abstract like "large" can take very specific dimensions in the form of "how large on a scale of 0 to 1" it is. The process of coding data into these states is called *fuzzification*.

Once all the data is coded in this manner, the various mappings are merged together through logical operators, such as inference rules (for example, "If A and B then C," where A and B correspond to states of two different features and C to the target variable). The result is a new membership function describing this complex relationship, usually depicted as a polygon. This is then turned into a crisp value, through one of various methods, in a process called *defuzzification*. Since this whole process is graphically accessible to the user, and the terms used

Copyrighted material

are borrowed from human language, the result is always something clear-cut and interpretable (given some understanding of how FL works).

Interestingly, FL has also been used in conjunction with ANNs to form what are referred to as *neuro-fuzzy* systems. Instead of having a person create the membership functions by hand, a FL system can make use of the optimization method in a neural network's training algorithm to calculate them on the fly. This whole process and the data structure that it entails take the form of an automated fuzzy system, combining the best of both worlds.

Why systems based on fuzzy logic are still relevant

Although FL was originally developed with a certain types of engineering systems in mind such as Control Systems, its ease of use and low cost of implementation has made it relevant as an AI methodology across a variety of other fields, including data science.

What's more, fuzzy systems are very accessible, especially when automated through optimization for their membership functions (such as the neuro-fuzzy systems mentioned previously). Such a system employs a set of FL rules (which are created based on the data) to infer the target variable. These systems are called Fuzzy Inference Systems, or FISs.

The main advantage of this FIS approach is that it is transparent—a big plus if you want to defend your results to the project stakeholders. The transparency of a FIS makes the whole problem more understandable, enabling you to figure out which features are more relevant.

In addition, a FIS can be used in conjunction with custom-made rules based on an expert's knowledge. This is particularly useful if you are looking at upgrading a set of heuristic rules using AI. Certain larger companies that are planning to use data science to augment their existing systems are likely to be interested in such a solution.

Downside of fuzzy inference systems

Copyrighted material

Despite all the merits of FIS, these AI systems don't always meet the expectations of modern data science projects. Specifically, when the dimensionality of the data at hand is quite large, the number of rules produced increases exponentially, making these systems too large for any practical purposes. Of course, you can mitigate this issue with an autoencoder or a statistical process, like PCA or ICA, that creates a smaller set of features. However, when you do this, the whole interpretability benefit of FIS goes out the window. Why? With a reduced feature set, the relationship with the original features (and the semantic meaning they carry) is warped. As such, it is very difficult to reconstruct meaning; the new features will require a different interpretation if they are to be meaningful. This is not always feasible.

Nevertheless, for datasets of smaller dimensionality, a FIS is a worthwhile alternative, even if it's not a particularly popular one. We'll explore Fuzzy Logic and FIS more in Chapter 11, where we'll discuss alternative AI methodologies.

Artificial creativity

Artificial creativity (AC) is a relatively new methodology of AI, where new information is created based on relevant data it has been trained on. Its applications span across various domains, including most of the arts, as well as industrial design, and even data science.

This kind of AI methodology makes use of a specialized DL network that is trained to develop new data that retains some characteristics of the data it was trained on. When you feed such a specialized AI system some image data, and it's been trained on the artwork of a particular painter, it will produce new "artwork" that makes use of the images it is fed, but using the painting patterns of the artist it is trained to emulate. The results may not win any art prizes, but they are certainly interesting and original!

It is particularly fascinating when an AC system creates poetry, based on the verse of certain poets. The results can be indistinguishable from human-written verse. Take for example the following piece of verse by

Copyrighted material

the AI poet Deep Gimble I:[6]

> *Madness in her face and i*
> *the world that I had seen*
> *and when my soul shall be to see the night to be the same and*
> *I am all the world and the day that is the same and a day I had been*
> *a young little woman I am in a dream that you were in*
> *a moment and my own heart in her face of a great world*
> *and she said the little day is a man of a little*
> *a little one of a day of my heart that has been in a dream.*

In data science, AC can aid in the creation of new data, which is quite useful in certain cases. This new data may not be particularly helpful as an expansion of the training set for that ANN, but it can be especially useful in other ways. For example, if the original data is sensitive like medical data, and contains too much *personally identifiable information (PII),* you can generate new data using AC that, although very similar to the original data, cannot be mapped back to a real individual.

In addition, data created from an AC system can be useful for different data models—perhaps as a new test set or even part of their training set. This way it can offer the potential for better generalization for these models, as there is more data available for them to train or test on. This can be particularly useful in domains where labeled data is hard to come by or is expensive to generate otherwise.

Additional AI methodologies

Beyond all the AI methodologies we've discussed so far, there exist several others worth noting. These systems also have a role to play in data science, while their similarities to DL systems make them easier to comprehend. Also, as the AI field is constantly expanding, it's good to be aware of all the new methodologies that pop up.

The Extreme Learning Machine (or ELM) is an example of an alternative AI methodology that hasn't yet received the attention it deserves. Although they are architecturally like DL networks, ELMs are quite distinct in the way they are trained. In fact, their training is so

Copyrighted material

unconventional that some people considered the whole approach borderline unscientific (the professor who came up with ELMs most recently has received a lot of criticism from other academics).

Instead of optimizing all the weights across the network, ELMs focus on just the connections of the two last layers—namely the last set of meta-features and their outputs. The rest of the weights maintain their initial random values from the beginning. Because the focus is solely on just the optimized weights of the last layers, this optimization is extremely fast and very precise.

As a result, ELMs are the fastest network-based methodology out there, and their performance is quite decent too. What's more, they are quite unlikely to overfit, which is another advantage. Despite its counter-intuitive approach, an ELM system does essentially the same thing as a conventional DL system; instead of optimizing all the meta-features it creates, though, it focuses on optimizing the way they work together to form a predictive analytics model. We'll talk more about ELMs in Chapter 11.

Another new alternative AI methodology is Capsule Networks (CapsNets). Although the CapsNet should be regarded as a member of the deep learning methods family, its architecture and its optimization training method are quite novel. CapsNets try to capture the relative relationships between the objects within a relevant context. A CNN model that achieves high performance in image recognition tasks may not necessarily be able to identify the same object from different angles. CapsNets, though, capture those kinds of contextual relationships quite well. Their performance on some tasks has already surpassed the leading models by about 45%, which is quite astonishing. Considering their promising future, we dedicate a section in Chapter 11 to discussing CapsNets.

Self-organizing Maps (SOMs) are a special type of AI system. Although they are also ANNs of sorts, they are unique in function. SOMs offer a way to map the feature space into a two-dimensional grid, so that it can be better visualized afterwards. Since it doesn't make use of a target variable, a SOM is an unsupervised learning methodology; as such, it is ideal for *data exploration*.

SOMs have been successfully applied in various domains, such as

Copyrighted material

meteorology, oceanography, oil and gas exploration, and project prioritization. One key difference SOMs have from other ANNs is that their learning is based on competition instead of error correction. Also, their architecture is quite different, as their various nodes are only connected to the input layer with no lateral connections. This unique design was first introduced by Professor Kohonen, which is why SOMs are also referred to as "Kohonen Maps."

The *Generative Adversarial Network (GAN)* is a very interesting type of AI methodology, geared towards optimizing a DL network in a rather creative way. A GAN comprises two distinct ANNs. One is for learning, and the other is for "breaking" the first one – finding cases where the predictions of the first ANN are off. These systems are comparable to the "white hat" hackers of cybersecurity.

In essence, the second ANN creates increasingly more demanding challenges for the first ANN, thereby constantly improving its generalization (even with a limited amount of data). GANs are used for simulations as well as data science problems. Their main field of application is astronomy, where a somewhat limited quantity of images and videos of the cosmos is available to use in training. The idea of GANs has been around for over a decade, but has only recently managed to gain popularity; this is largely due to the amount of computational resources demanded by such a system (just like any other DL-related AI system). A more detailed description of GANs is available in Appendix D, while in Chapter 9 we'll also revisit this topic.

Artificial Emotional Intelligence (AEI) is another kind of AI that's novel on both the methodological as well as the application levels. The goal of AEI is to facilitate an understanding of the emotional context of data (which is usually text-based) and to assess it just like a human. Applications of AEI are currently limited to comprehension; in the future, though, more interactive systems could provide a smoother interface between humans and machines. There is an intersect between AEI and ANNs, but some aspects of AEI make use of other kinds of ML systems on the back end.

Glimpse into the future

Copyrighted material

While the field of AI expands in various directions, making it hard to speculate about how it will evolve, there is one common drawback to most of the AI systems used today: a lack of interpretability. As such, it is quite likely that some future AI system will address this matter, providing a more comprehensive result, or at least some information as to how the result came about (something like a rationale for each prediction), all while maintaining the scalability of modern AI systems.

A more advanced AI system of the future will likely have a network structure, just like current DL systems—though it may be quite different architecturally. Such a system would be able to learn with fewer data points (possibly assisted by a GAN), as well as generate new data (just like variational autoencoders).

Could an AI system learn to build new AI systems? It is possible, however the limitation of excessive resources required for such a task has made it feasible only for cloud-based systems. Google may showcase its progress in this area in what it refers to as *Automated Machine Learning (AutoML)*.

So, if you were to replicate this task with your own system, who is to say that the AI-created AI would be better than what you yourself would have built? Furthermore, would you be able to pinpoint its shortcomings, which may be quite subtle and obscure? After all, an AI system requires a lot of effort to make sure that its results are not just accurate but also useful, addressing the end-user's needs. You can imagine how risky it would be to have an AI system built that you know nothing about!

However, all this is just an idea of a potential evolutionary course, since AI can always evolve in unexpected ways. Fortunately, with all the popularity of AI systems today, if something new and better comes along, you'll probably find out about it sooner rather than later.

Perhaps for things like that it's best to stop and think about the why's instead of focusing only on the how's, since as many science and industry experts have warned us, AI is a high-risk endeavor and needs to be handled carefully and always with fail-safes set in place. For example, although it's fascinating and in some cases important to think about how we can develop AIs that improve themselves, it's also crucial to understand what implications this may have and plan for AI safety

Copyrighted material

matters beforehand. Also, prioritizing certain characteristics of an AI system (e.g. interpretability, ease of use, having limited issues in the case of malfunction, etc.) over raw performance, may provide more far-reaching benefits. After all, isn't improving our lives in the long-term the reason why we have AI in the first place?

About the methods

It is not hard to find problems that can be tackled with optimization. For example, you may be looking at an optimum configuration of a marketing process to minimize the total cost, or to maximize the number of people reached. Although data science can lend some aid in solving such a problem, at the end of the day, you'll need to employ an optimizer to find a true solution to a problem like this.

Furthermore, optimization can help in data engineering, too. Some feature selection methods, for instance, use optimization to keep only the features that work well together. There are also cases of *feature fusion* that employ optimization (although few people use this method, since it sacrifices some interpretability of the model that makes use of these meta-features).

In addition, when building a custom predictive analytics system combining other *classifiers* or *regressors*, you often need to maximize the overall accuracy rate (or some other performance metric). To do this, you must figure out the best parameters for each module (i.e. the ones that optimize a certain performance metric for the corresponding model), and consider the weights of each module's output in the overall decision rule for the final output of the system that comprises of all these modules. This work really requires an optimizer, since often the number of variables involved is considerable.

In general, if you are tackling a predictive analytics problem and you have a dataset whose dimensionality you have reduced through feature selection, it can be effectively processed through a FIS. In addition, if interpretability is a key requirement for your data model, using a FIS is a good strategy to follow. Finally, if you already have a set of heuristic rules at your disposal from an existing predictive analytics system, then

Copyrighted material

you can use a FIS to merge those rules with some new ones that the FIS creates. This way, you won't have to start from square one when developing your solution.

Novel AI systems tend to be less predictable and, as a result, somewhat unreliable. They may work well for a certain dataset, but that performance may not hold true with other datasets.

That's why it is critical to try out different AI systems before settling on one to use as your main data model. In many cases, optimizing a certain AI system may yield better performance, despite the time and resources it takes to optimize. Striking the balance between exploring various alternatives and digging deeper into existing ones is something that comes about with experience.

Moreover, it's a good idea to set your project demands and user requirements beforehand. Knowing what is needed can make the selection of your AI system (or if you are more adventurous, your design of a new one) much easier and more straightforward. For example, if you state early on that interpretability is more important than performance, this will affect which model you decide to use. Make sure you understand what you are looking for in an AI system from the beginning, as this is bound to help you significantly in making the optimal choice.

Although AI systems have a lot to offer in all kinds of data science problems, they are not panaceas. If the data at your disposal is not of high veracity, meaning not of good quality or reliability, no AI system can remedy that. All AI systems function based on the data we train them with; if the training data is very noisy, biased, or otherwise problematic, their generalizations are not going to be any better.

This underlines the importance of data engineering and utilizing data from various sources, thereby maximizing your chances of creating a robust and useful data model. This is also why it's always good to always keep a human in the loop when it comes to data science projects —even (or perhaps especially) when they evolve AI.

Summary

Copyrighted material

- Optimization is an AI-related process for finding the maximum or minimum of a given function, by tweaking the values for its variables. It is an integral part of many other systems (including ANNs) and consists of deterministic and stochastic systems.
- Optimization systems (or "optimizers," as they are often called) can be implemented in all programming languages, since their main algorithms are fairly straightforward.
- Fuzzy Logic (FL) is an AI methodology that attempts to model imprecise data as well as uncertainty. Systems employing FL are referred to as Fuzzy Inference Systems (FIS). These systems involve the development and use of fuzzy rules, which automatically link features to the target variable. A FIS is great for datasets of small dimensionality, since it doesn't scale well as the number of features increases.
- Artificial creativity (AC) is an AI methodology of sorts that creates new information based on patterns derived from the data it is fed. It has many applications in the arts and industrial design. This methodology could also be useful in data science, through the creation of new data points for sensitive datasets (for example, where data privacy is important).
- Artificial Emotional Intelligence (AEI) is another alternative AI, emulating human emotions. Currently its applications are limited to comprehension.
- Although speculative, if a truly novel AI methodology were to arise in the near future, it would probably combine the characteristics of existing AI systems, with an emphasis on interpretability.
- Even though theoretically possible, an AI system that can design and build other AI systems is not a trivial task. A big part of this involves the excessive risks of using such a system, since we would have little control over the result.
- It's important to understand the subtle differences between all these methodologies, as well as their various limitations. Most

Copyrighted material

importantly, for data science, there is no substitute for high veracity data.

http://bit.ly/2Js4CKd.

Copyrighted material

CHAPTER 3

Building a DL Network Using MXNet

We'll begin our in-depth examinations of the DL frameworks with that which seems one of the most promising: Apache's MXNet. We'll cover its core components including the *Gluon* interface, *NDArrays*, and the MXNet package in Python. You will learn how you can save your work like the networks you trained in data files, and some other useful things to keep in mind about MXNet.

MXNet supports a variety of programming languages through its API, most of which are useful for data science. Languages like Python, Julia, Scala, R, Perl, and C++ have their own wrappers of the MXNet system, which makes them easily integrated with your pipeline.

Also, MXNet allows for parallelism, letting you take full advantage of your machine's additional hardware resources, such as extra CPUs and GPUs. This makes MXNet quite fast, which is essential when tackling computationally heavy problems, like the ones found in most DL applications.

Interestingly, the DL systems you create in MXNet can be deployed on all kinds of computer platforms, including smart devices. This is possible through a process called *amalgamation*, which ports a whole system into a single file that can then be executed as a standalone program. Amalgamation in MXNet was created by Jack Deng, and involves the development of .cc files, which use the BLAS library as their only dependency. Files like this tend to be quite large (more than 30000 lines long). There is also the option of compiling .h files using a program called *emscripten*. This program is independent of any library, and can be used by other programming languages with the corresponding API.

Finally, there exist several tutorials for MXNet, should you wish to learn more about its various functions. Because MXNet is an open-source project, you can even create your own tutorial, if you are so inclined. What's more, it is a

Copyrighted material

cross-platform tool, running on all major operating systems. MXNet has been around long enough that it is a topic of much research, including a well-known academic paper by Chen et al.[7]

Core components

Gluon interface

Gluon is a simple interface for all your DL work using MXNet. You install it on your machine just like any Python library:

```
pip install mxnet --pre --user
```

The main selling point of Gluon is that it is straightforward. It offers an abstraction of the whole network building process, which can be intimidating for people new to the craft. Also, Gluon is very fast, not adding any significant overhead to the training of your DL system. Moreover, Gluon can handle dynamic graphs, offering some malleability in the structure of the ANNs created. Finally, Gluon has an overall flexible structure, making the development process for any ANN less rigid.

Naturally, for Gluon to work, you must have MXNet installed on your machine (although you don't need to if you are using the Docker container provided with this book). This is achieved using the familiar pip command:

```
pip install mxnet --pre --user
```

Because of its utility and excellent integration with MXNet, we'll be using Gluon throughout this chapter, as we explore this DL framework. However, to get a better understanding of MXNet, we'll first briefly consider how you can use some of its other functions (which will come in handy for one of the case studies we examine later).

NDArrays

The NDArray is a particularly useful *data structure* that's used throughout an

Copyrighted material

MXNet project. NDArrays are essentially NumPy arrays, but with the added capability of asynchronous CPU processing. They are also compatible with distributed cloud architectures, and can even utilize automatic differentiation, which is particularly useful when training a deep learning system, but NDArrays can be effectively used in other ML applications too. NDArrays are part of the MXNet package, which we will examine shortly. You can import the NDArrays module as follows:

```
from mxnet import nd
```

To create a new NDArray consisting of 4 rows and 5 columns, for example, you can type the following:

```
nd.empty((4, 5))
```

The output will differ every time you run it, since the framework will allocate whatever value it finds in the parts of the memory that it allocates to that array. If you want the NDArray to have just zeros instead, type:

```
nd.zeros((4, 5))
```

To find the number of rows and columns of a variable having an NDArray assigned to it, you need to use the *.shape* function, just like in NumPy:

```
x = nd.empty((2, 7))
x.shape
```

Finally, if you want to find to total number of elements in an NDArray, you use the *.size* function:

```
x.size
```

The operations in an NDArray are just like the ones in NumPy, so we won't elaborate on them here. Contents are also accessed in the same way, through indexing and slicing.

Should you want to turn an NDArray into a more familiar data structure from the NumPy package, you can use the *asnumpy()* function:

```
y = x.asnumpy()
```

Copyrighted material

The reverse can be achieved using the *array()* function:

```
z = nd.array(y)
```

One of the distinguishing characteristics of NDArrays is that they can assign different computational contexts to different arrays—either on the CPU or on a GPU attached to your machine (this is referred to as "*context*" when discussing about NDArrays). This is made possible by the *ctx* parameter in all the package's relevant functions. For example, when creating an empty array of zeros that you want to assign to the first GPU, simply type:

```
a = nd.zeros(shape=(5,5), ctx=mx.gpu(0))
```

Of course, the data assigned to a particular processing unit is not set in stone. It is easy to copy data to a different location, linked to a different processing unit, using the *copyto()* function:

```
y = x.copyto(mx.gpu(1)) # copy the data of NDArray x to
the 2nd GPU
```

You can find the context of a variable through the *.context* attribute:

```
print(x.context)
```

It is often more convenient to define the context of both the data and the models, using a separate variable for each. For example, say that your DL project uses data that you want to be processed by the CPU, and a model that you prefer to be handled by the first GPU. In this case, you'd type something like:

```
DataCtx = mx.cpu()
ModelCtx = mx.gpu(0)
```

MXNet package in Python

The MXNet package (or "mxnet," with all lower-case letters, when typed in Python), is a very robust and self-sufficient library in Python. MXNet provides deep learning capabilities through the MXNet framework. Importing this package in Python is fairly straightforward:

Copyrighted material

```
import mxnet as mx
```

If you want to perform some additional processes that make the MXNet experience even better, it is highly recommended that you first install the following packages on your computer:

- *graphviz* (ver. 0.8.1 or later)
- *requests* (ver. 2.18.4 or later)
- *numpy* (ver. 1.13.3 or later)

You can learn more about the MXNet package through the corresponding GitHub repository.[8]

MXNet in action

Now let's take a look at what we can do with MXNet, using Python, on a Docker image with all the necessary software already installed. We'll begin with a brief description of the datasets we'll use, and then proceed to a couple specific DL applications using that data (namely classification and regression). Upon mastering these, you can explore some more advanced DL systems of this framework on your own.

Datasets description

In this section we'll introduce two synthetic datasets that we prepared to demonstrate classification and regression methods on them. First dataset is for classification, and the other for regression. The reason we use synthetic datasets in these exercises to maximize our understanding of the data, so that we can evaluate the results of the DL systems independent of data quality.

The first dataset comprises 4 variables, 3 features, and 1 labels variable. With 250,000 data points, it is adequately large for a DL network to work with. Its small dimensionality makes it ideal for visualization (see Figure 2). It is also made to have a great deal of non-linearity, making it a good challenge for any data model (though not too hard for a DL system). Furthermore, classes 2 and 3 of this dataset are close enough to be confusing, but still distinct. This makes

Copyrighted material

them a good option for a clustering application, as we'll see later.

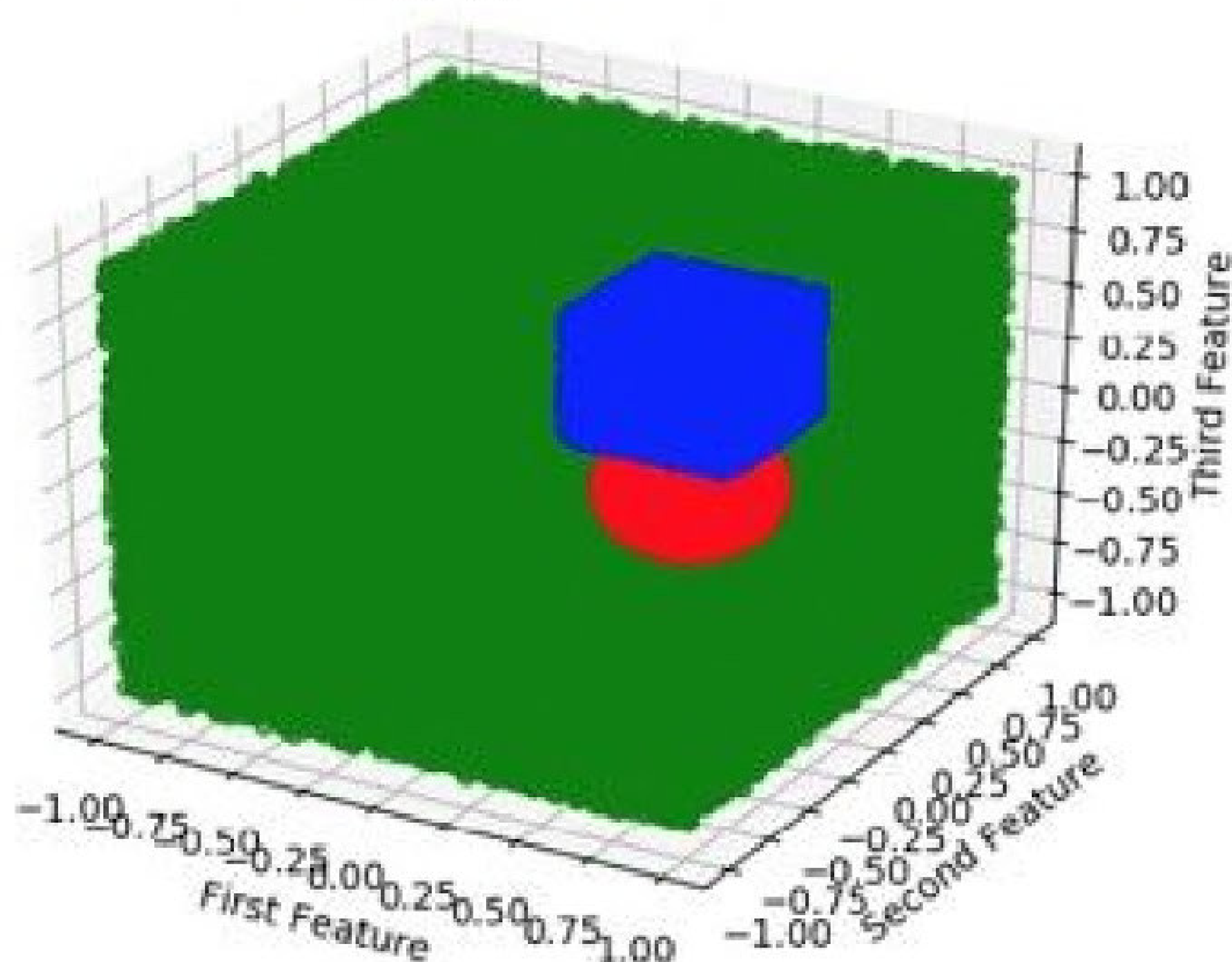

Figure 2: A graphical representation of the first dataset to be used in these examples. The different colors correspond to the different classes of the dataset. Although the three classes are defined clearly, figuring out in which one of them an unknown data point belongs is non-linear. This is due to the presence of the spherical class, making the whole problem challenging enough to render a DL approach to it relevant.

The second dataset is somewhat larger, comprising 21 variables—20 of which are the features used to predict the last, which is the target variable. With 250,000 data points, again, it is ideal for a DL system. Note that only 10 of the 20 features are relevant to the target variable (which is a combination of these 10). A bit of noise is added to the data to make the whole problem a bit more challenging. The remaining 10 features are just random data that must be filtered out by the DL model. Relevant or not, this dataset has enough features altogether to render a dimensionality reduction application worthwhile. Naturally, due to its dimensionality, we cannot plot this dataset.

Loading a dataset into an NDArray

Let's now take a look at how we can load a dataset in MXNet, so that we can

Copyrighted material

process it with a DL model later on. First let's start with setting some parameters:

```
DataCtx = mx.cpu() # assign context of the data used
BatchSize = 64 # batch parameter for dataloader object
r = 0.8 # ratio of training data
nf = 3 # number of features in the dataset (for the
classification problem)
```

Now, we can import the data like we'd normally do in a conventional DS project, but this time store it in NDArrays instead of Pandas or NumPy arrays:

```
with open("../data/data1.csv") as f:
    data_raw = f.read()
lines = data_raw.splitlines() # split the data into
separate lines
ndp = len(lines) # number of data points
X = nd.zeros((ndp, nf), ctx=data_ctx)
Y = nd.zeros((ndp, 1), ctx=data_ctx)
for i, line in enumerate(lines):
tokens = line.split()
Y[i] = int(tokens[0])
for token in tokens[1:]:
    index = int(token[:-2]) - 1
    X[i, index] = 1
```

Now we can split the data into a *training set* and a *testing set*, so that we can use it both to build and to validate our classification model:

```
import numpy as np # we'll be needing this package as
well
n = np.round(N * r) # number of training data points
train = data[:n, ] # training set partition
test = data[(n + 1):,] # testing set partition
data_train =
gluon.data.DataLoader(gluon.data.ArrayDataset(train[:,:3],
train[:,3]), batch_size=BatchSize, shuffle=True)
data_test =
gluon.data.DataLoader(gluon.data.ArrayDataset(test[:,:3],
test[:,3]), batch_size=BatchSize, shuffle=True)
```

Copyrighted material

We'll then need to repeat the same process to load the second dataset—this time using *data2.csv* as the source file. Also, to avoid confusion with the dataloader objects of dataset 1, you can name the new dataloaders *data_train2* and *data_test2*, respectively.

Classification

Now let's explore how we can use this data to build an MLP system that can discern the different classes within the data we have prepared. For starters, let's see how to do this using the mxnet package on its own; then we'll examine how the same thing can be achieved using Gluon.

First, let's define some constants that we'll use later to build, train, and test the MLP network:

```
nhn = 256 # number of hidden nodes for each layer
WeightScale = 0.01 # scale multiplier for weights
ModelCtx = mx.cpu() # assign context of the model itself
no = 3 # number of outputs (classes)
ne = 10 # number of epochs (for training)
lr = 0.001 # learning rate (for training)
sc = 0.01 # smoothing constant (for training)
ns = test.shape[0] # number of samples (for testing)
```

Next, let's initialize the network's parameters (weights and biases) for the first layer:

```
W1 = nd.random_normal(shape=(nf, nhn),
scale=WeightScale, ctx=ModelCtx)
b1 = nd.random_normal(shape=nhn, scale=WeightScale,
ctx=ModelCtx)
```

And do the same for the second layer:

```
W2 = nd.random_normal(shape=(nhn, nhn),
scale=WeightScale, ctx=ModelCtx)
b2 = nd.random_normal(shape=nhn, scale=WeightScale,
ctx=ModelCtx)
```

Copyrighted material

Then let's initialize the output layer and aggregate all the parameters into a single data structure called *params*:

```
W3 = nd.random_normal(shape=(nhn, no),
scale=WeightScale, ctx=ModelCtx)
b3 = nd.random_normal(shape=no, scale=WeightScale,
ctx=ModelCtx)
params = [W1, b1, W2, b2, W3, b3]
```

Finally, let's allocate some space for a gradient for each one of these parameters:

```
for param in params:
    param.attach_grad()
```

Remember that without any non-linear functions in the MLP's neurons, the whole system would be too rudimentary to be useful. We'll make use of the *ReLU* and the *Softmax* functions as activation functions for our system:

```
def relu(X): return nd.maximum(X, nd.zeros_like(X))
def softmax(y_linear):
    exp = nd.exp(y_linear - nd.max(y_linear))
    partition = nd.nansum(exp, axis=0,
exclude=True).reshape((-1, 1))
    return exp / partition
```

Note that the Softmax function will be used in the output neurons, while the ReLU function will be used in all the remaining neurons of the network.

For the cost function of the network (or, in other words, the fitness function of the optimization method under the hood), we'll use the cross-entropy function:

```
def cross_entropy(yhat, y): return - nd.nansum(y *
nd.log(yhat), axis=0, exclude=True)
```

To make the whole system a bit more efficient, we can combine the softmax and the cross-entropy functions into one, as follows:

```
def softmax_cross_entropy(yhat_linear, y):
```

Copyrighted material

```
    return - nd.nansum(y * nd.log_softmax(yhat_linear),
axis=0, exclude=True)
```

After all this, we can now define the function of the whole neural network, based on the above architecture:

```
def net(X):
    h1_linear = nd.dot(X, W1) + b1
    h1 = relu(h1_linear)
    h2_linear = nd.dot(h1, W2) + b2
    h2 = relu(h2_linear)
    yhat_linear = nd.dot(h2, W3) + b3
    return yhat_linear
```

The optimization method for training the system must also be defined. In this case we'll utilize a form of Gradient Descent:

```
def SGD(params, lr):
for param in params:
        param[:] = param - lr * param.grad
return param
```

For the purposes of this example, we'll use a simple evaluation metric for the model: accuracy rate. Of course, this needs to be defined first:

```
def evaluate_accuracy(data_iterator, net):
    numerator = 0.
    denominator = 0.
    for i, (data, label) in enumerate(data_iterator):
        data =
data.as_in_context(model_ctx).reshape((-1, 784))
        label = label.as_in_context(model_ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        numerator += nd.sum(predictions == label)
        denominator += data.shape[0]
    return (numerator / denominator).asscalar()
```

Now we can train the system as follows:

Copyrighted material

```
for e in range(epochs):
    cumulative_loss = 0
    for i, (data, label) in enumerate(train_data):
        data =
data.as_in_context(model_ctx).reshape((-1, 784))
        label = label.as_in_context(model_ctx)
        label_one_hot = nd.one_hot(label, 10)
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output,
label_one_hot)
        loss.backward()
        SGD(params, learning_rate)
        cumulative_loss += nd.sum(loss).asscalar()
    test_accuracy = evaluate_accuracy(test_data, net)
    train_accuracy = evaluate_accuracy(train_data, net)
    print("Epoch %s. Loss: %s, Train_acc %s, Test_acc
%s" %
          (e, cumulative_loss/num_examples,
train_accuracy, test_accuracy))
```

Finally, we can use to system to make some predictions using the following code:

```
def model_predict(net, data):
output = net(data)
return nd.argmax(output, axis=1)
SampleData = mx.gluon.data.DataLoader(data_test, ns,
shuffle=True)
for i, (data, label) in enumerate(SampleData):
data = data.as_in_context(ModelCtx)
im = nd.transpose(data,(1,0,2,3))
im = nd.reshape(im,(28,10*28,1))
imtiles = nd.tile(im, (1,1,3))
plt.imshow(imtiles.asnumpy())
plt.show()
pred=model_predict(net,data.reshape((-1,784)))
print('model predictions are:', pred)
print('true labels :', label)
break
```

Copyrighted material

If you run the above code (preferably in the Docker environment provided), you will see that this simple MLP system does a good job at predicting the classes of some unknown data points—even if the class boundaries are highly non-linear. Experiment with this system more and see how you can improve its performance even further, using the MXNet framework.

Now we'll see how we can significantly simplify all this by employing the Gluon interface. First, let's define a Python class to cover some common cases of Multi-Layer Perceptrons, transforming a "gluon.Block" object into something that can be leveraged to gradually build a neural network, consisting of multiple layers (also known as MLP):

```
class MLP(gluon.Block):
def __init__(self, **kwargs):
super(MLP, self).__init__(**kwargs)
  with self.name_scope():
  self.dense0 = gluon.nn.Dense(64) # architecture of 1st
layer (hidden)
  self.dense1 = gluon.nn.Dense(64) # architecture of 2nd
layer (hidden)
  self.dense2 = gluon.nn.Dense(3) # architecture of 3rd
layer (output)
  def forward(self, x): # a function enabling an MLP to
process data (x) by passing it forward (towards the
output layer)
    x = nd.relu(self.dense0(x)) # outputs of first
hidden layer
    x = nd.relu(self.dense1(x)) # outputs of second
hidden layer
    x = self.dense2(x) # outputs of final layer (output)
    return x
```

Of course, this is just an example of how you can define an MLP using Gluon, not a one-size-fits-all kind of solution. You may want to define the MLP class differently, since the architecture you use will have an impact on the system's performance. (This is particularly true for complex problems where additional hidden layers would be useful.) However, if you find what follows too challenging, and you don't have the time to assimilate the theory behind DL systems covered in Chapter 1, you can use an MLP object like the one

Copyrighted material

above for your project.

Since DL systems are rarely as compact as the MLP above, and since we often need to add more layers (which would be cumbersome in the above approach), it is common to use a different class called *Sequential*. After we define the number of neurons in each hidden layer, and specify the activation function for these neurons, we can build an MLP like a ladder, with each step representing one layer in the MLP:

```
nhn = 64 # number of hidden neurons (in each layer)
af = "relu" # activation function to be used in each
neuron
net = gluon.nn.Sequential()
with net.name_scope():
net.add(gluon.nn.Dense(nhn , activation=af))
net.add(gluon.nn.Dense(nhn , activation=af))
net.add(gluon.nn.Dense(no))
```

This takes care of the architecture for us. To make the above network functional, we'll first need to initialize it:

```
sigma = 0.1 # sigma value for distribution of weights
for the ANN connections
ModelCtx = mx.cpu()
lr = 0.01 # learning rate
oa = 'sgd' # optimization algorithm
net.collect_params().initialize(mx.init.Normal(sigma=sigma),
ctx=ModelCtx)
softmax_cross_entropy =
gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), oa,
{'learning_rate': lr})
ne = 10 # number of epochs for training
```

Next, we must define how we assess the network's progress, through an evaluation metric function. For the purposes of simplicity, we'll use the standard accuracy rate metric:

```
def AccuracyEvaluation(iterator, net):
    acc = mx.metric.Accuracy()
    for i, (data, label) in enumerate(iterator):
```

Copyrighted material

```
        data = data.as_in_context(ModelCtx).reshape((-1,
3))
        label = label.as_in_context(ModelCtx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

Finally, it's time to train and test the MLP, using the aforementioned settings:

```
for e in range(ne):
    cumulative_loss = 0
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ModelCtx).reshape((-1,
784))
        label = label.as_in_context(ModelCtx)
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output, label)
        loss.backward()
        trainer.step(data.shape[0])
        cumulative_loss += nd.sum(loss).asscalar()
    train_accuracy = AccuracyEvaluation(train_data, net)
    test_accuracy = AccuracyEvaluation(test_data, net)
    print("Epoch %s. Loss: %s, Train_acc %s, Test_acc
%s" %
          (e, cumulative_loss/ns, train_accuracy,
test_accuracy))
```

Running the above code should yield similar results to those from conventional mxnet commands.

To make things easier, we'll rely on the Gluon interface in the example that follows. Nevertheless, we still recommend that you experiment with the standard mxnet functions afterwards, should you wish to develop your own architectures (or better understand the theory behind DL).

Regression

Copyrighted material