UNIVERSITÀ DEGLI STUDI
DI TORINO

FACOLTÀ DI SCIENZE
MATEMATICHE FISICHE E NATURALI

A Deep Learning Model to Forecast Financial Time-Series

Dissertazione di Laurea Magistrale in Fisica dei Sistemi Complessi

Relatore:
Prof. Pietro Terna

Candidato:
Gabriele D'Acunto

Controrelatore:
Prof. Michele Caselle

Anno Accademico 2015/2016

L'artista dopo che ha lavorato
deve sentirsi stanco, eccitato, qualche volta
felice e quasi sempre insoddisfatto.
$\sim$ Giacomo Balla

Introduction

Intelligent machines have long been dreamt by inventors. Indeed this desire dates back to at least the time of ancient Greece: several mythical figures such as Daedalus, Hephaestus and Palamades may all be interpreted as legendary inventors, Talos and Pandora may be instances of artificial life. In recent years *artificial intelligence* (AI) has become a prosperous field, providing practical solutions to automate routine labour, recognizing speeches or images and supporting scientific researches such as the ones in medical field.
The great success achieved by intelligent algorithms is mainly due to the fact that computers are able to counterbalance and compensate human incapability to formalize problems. Indeed everything that could be formalized trough a list of instructions represent a straight-forward and easy task for computers.
Intelligent models learn from experience and understand the world in terms of a hierarchy of concepts. In fact they learn concepts starting from easier ones. If we draw a graph showing how hierarchy works, we would see it is *deep*, with many layers. This is the reason why we call this approach *deep learning*. By gathering knowledge from experience, this procedure avoids human operators to formally specify all of the instructions that the model needs.
In this thesis we propose a deep learning model to forecast financial time-series.
Every trader dreams of discovering a flawless method to forecast the trend of stock market. This procedure would allow investors to invest in a safe way as well as to gain large sums of money. Clearly those who eventually find out about this method should be careful in bringing it to light since, as every secret revelation, it would lose its magic.
There are two main practices used to face stock market:

- *Fundamental Analysis*: it tries to select the "right price" of a specific financial quantity (for instance a share or an index) resorting to the study of economy dynamics, balance sheet data, movement of interest rates and so on. In other words, this analysis starts from the *causes* in order to prevent the *effects*, thus changes in prices. If today price is lower than the theoretical estimated one investors buy, otherwise they sell. However, we have to say it is very hard to identify and especially to correctly evaluate all the relevant factors, even with the adoption of adequate econometric models. Indeed, the main problem of this approach is related to two assumptions: investors have rational expectations and the market is efficient; instead *rumors* are often the motive of fluctuations in prices. Moreover, during data evaluation it is fundamental to keep in mind that the market tends to anticipate outcomes; therefore, for example, if positive stock data are published the price immediately goes down, since it previously increased due to this moving up effect of the market. Here-hence the empirical rule: *buy on rumors, sell on facts.*

- *Technical Analysis*: unlike fundamental analysis it establishes that the "right price" related to a financial quantity is impossible to identify. This is the reason why technical analysts believe past trading activity and price changes of a security are better indicators of future price movements rather than the intrinsic value. Besides, the market is not assumed to be efficient and investors are not rational. Thus, this analysis does not aim at understanding market movements, but how to be at the right time in the right place to minimize losses and maximize utilities. Indeed it focuses on charts of prices movement and various analytical tools to evaluate a security's strength or weakness and forecast future price changes. Finally, this study allows to characterize entry and exit stock market levels, which are convenient as far as *risk-reward* is concerned, and to provide the exact and precise moment to act, known as *timing*.

The two instruments described above can be used in a complementary way, in particular fundamental analysis is preferred for long horizons, while technical analysis for the short-term. However this last one has been object of arguments among professionals about its reliability. Even though it is widely used among traders and financial professionals, there are evidences of a scarce success. Indeed academics, such as Eugene Fama, say the proof for technical analysis is inconsistent with the weak form of the *Efficient Market Hypothesis* (EMH). This one implies that the market reflects all its information and assumes that past rates of return have no effect of future rates. Given this assumption, rules such as the ones traders use to buy or sell a stock are invalid. Moreover users believe that, even if this study cannot predict future, it is useful to identify trends, tendencies and trading opportunities.
In this thesis we apply a third approach, different from the previous ones. In fact we do not define a strategy: we let the machine establishing what is relevant in order to forecast future closing prices. This model works starting from a finite number k of $x_k(t)$ time-series: from this a $h(t)$ financial scenario is defined and used to forecast a specific quantity $\hat{y}(t+i)$.
In particular, the development of this model is explained following the steps described thereafter.

1. Deep learning history is delineated in **Chapter 1** highlighting how this model roots in the past century. It started in 1940s with the movement of *cybernetics* but it was discredited because of the limits of first algorithms, also known as *linear classifiers*. In 1980s it reappeared with the *connectionism*, inspired to the studies of the psychologist Donald Hebb. However, only from 2006 deep learning has achieved large successes thanks to two key aspects: availability of large datasets and computational strength of modern technologies.

2. **Chapter 2** introduces the software used to develop the previous mentioned model in Python: *TensorFlow*, a machine learning library developed by *Google*

Brain team and open-sourced in the late 2015. The analysis is conducted disclosing its advantages and its basic concepts making use of short codes. Moreover we also describe TensorBoard, a graphical tool showing tensors flow charts, very helpful to visualize the structure of a model, to deeply understand what happens when running a code and eventually to debug it.

3. **Chapter 3** deals with the re-elaboration of three classic deep learning applications to MNIST (*Mixed National Institute of Standards and Technology*) dataset, made of handwritten digits form 0 to 9, commonly used for training and testing in the field of machine learning. More precisely, starting from the very beginning we develop a *Multilayer Perceptron* (MLP), a *Convolutional Neural Network* (CNN) and last but not least a *Recurrent Neural Network* (RNN) with *Long Short-Term Memory* cells (LSTM). Moreover, the way of programming used in this dissertation is introduced to the reader, who approaches some key elements of the proposed architecture such as *depth* and the implementation of LSTM, besides understanding how to build a deep learning model.
   We invite the reader to pay specific attention to contents of this chapter since it will be fundamental later on.

4. The development of the model is explained in **Chapter 4**: first of all the single parts composing it are deeply theoretically illustrated, then they are implemented. Despite the fact that the programming language is easy to understand, (unequivocal Python's strong point),it is also matched with detailed comments and remarkable key steps that show how to build the machine. Since this one is a deep architecture composed by two consecutive parts each representing a deep model it extends the concept of *depth*. The first one is a *deep autoencoder* whose task is to create the financial scenario from which the second part, i.e. a RNN with LSTM cells, forecasts the future values of a quantity of interest. Furthermore we present the *Adaptive moments* optimizer (ADAM) used in the implementation of the machine and we analyse the long-term dependencies problem of RNNs that has implied the development of new technologies such as the LSTM cell.

5. Without a doubt **Chapter 5** represents the most interesting as well as emblematic part of this thesis: the application of the deep learning model to financial time-series and subsequently the analysis of results.
   Here the main strengths and weak points of the proposed machine are bared and it is shown how the unforeseen and the unexpected have changed and guided this research towards different types of analysis. Starting from the forecast of **Toyota Motor Corporation** closing prices, we move to stationary series analysis(returns and lagged differences), until reaching a qualitative study, the most significant result of this research. Indeed we generalized this last approach to five securities belonging to NASDAQ-100 index: **Apple Inc.**, **Microsoft**, **Texas Instruments Incorporated**, **Intel Corporation**, **Adobe Systems Incorporated**. Other stocks belonging to the same index were discarded because of their small size. We finally provide an answer to

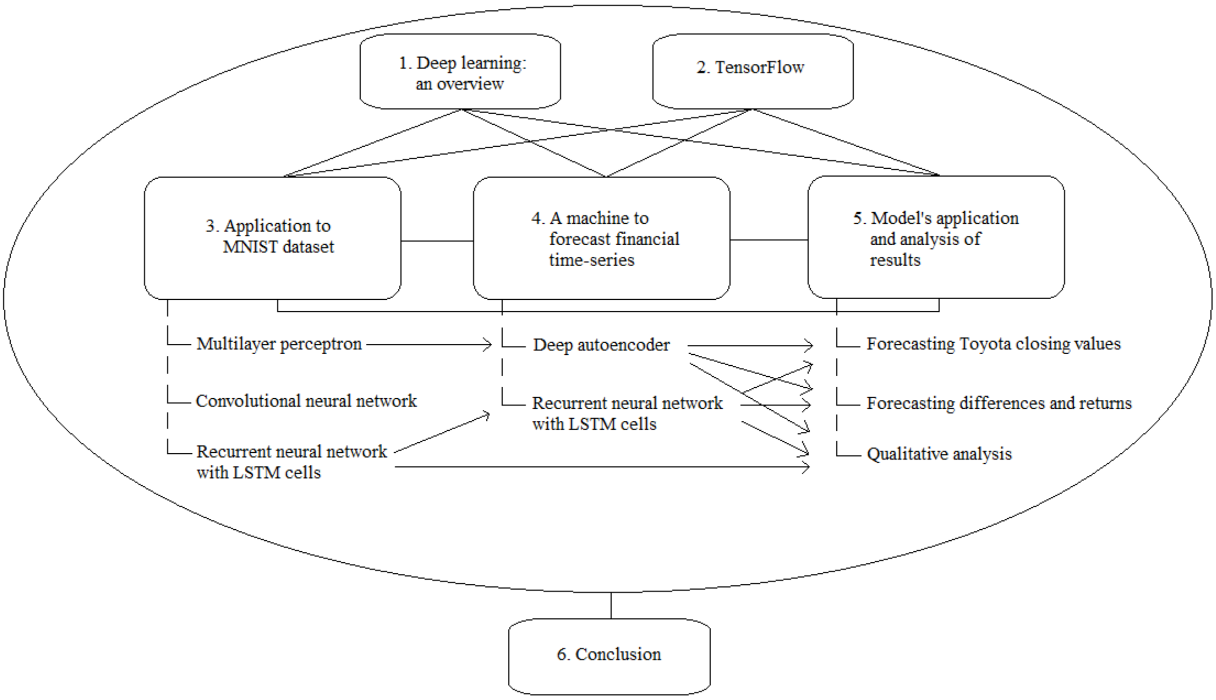


Figure 1: High-level organization of the thesis. A link (arrow) between two chapters (sections) indicates that the previous chapter is prerequisite material for understanding the following. In figure we only show the suitable sections to highlight connections among Chapters 3, 4 and 5. We use the big oval set to point out Chapter 6 is linked to every previous chapter.

how the model should be used for trading.
In particular, the research material related to *Recurrent Neural Network* (RNN) applied to MNIST database, presented in section 3.5, was very useful in order to fulfil this last approach.

6. We conclude this work with **Chapter 6** where we summarize the results achieved, highlighting both positive and negative aspects. Besides we provide hints for future developments indicating the additional studies we are currently following.

The financial data used in this dissertation were downloaded from `finance.yahoo.com`. Therefore during this research we face difficulties in collecting suitable and free material and in organizing it to create datasets for the proposed machine. Besides we display the Python codes to built these datasets as well.

Contents

Chapter 1

Deep learning: an overview

It is well known computers do not simply solve complicated equations, they appear to do a lot more. The problems we face seem often plenty of intricate details. Thus a traditional approach suggests to get a large knowledge about them in order to formalize and subsequently work them out. For instance consider converting speech to text: the aforementioned approach may involve understanding the human vocal chords to decipher utterances, knowing a specific lexicon with its own rule, using many hand-designed, domain-specific, un-generalized pieces of code. If in recent years programming allows us to automate tasks and save time, is because a programmer no longer needs to know each detailed part of a problem to solve it. It does not regard magic: it is possible to write codes that, given enough time and looking at enough examples, figure out a solution for a specific task.
The capability to acquire their own knowledge, by extracting patterns from data is named *machine learning*. Simple examples of machine learning algorithms are *logistic regression* and *naive Bayes*:

- the first one can establish for instance whether to recommend cesarean delivery. The machine does not analyse the patient, the doctor provides it important pieces of information (of course "important" for the computer). Each piece of information, involved in the characterization of the patient, represents a *feature*. Logistic regression evaluates how several features correlates between them, giving an outcome out of many others. Technically speaking the algorithm solves a *multilabel classification problem*;

- the *naive Bayes* is able to filter spam emails, separating them from the legitimate ones. Even this algorithm solves a *classification problem*. Indeed, it is a probabilistic classifier (relying on Bayes Theorem) that minimizes the risk of classification, considering the *features* one independent from the other.

In both instances presented above, data representation is crucial. Consider to pass to the logistic regression an MRI scan of the patient instead of the doctor report: the algorithm would not be able to carry out its task because the pixels of the scan have negligible correlation with any complications that might happen during delivery.
Choosing the right representation is a challenge not only in machine learning field, indeed we continuously face it during daily life: people can perform faster an operation using Arabic numerals rather than Roman ones. However the representation

Figure 1.1: "The Pirate" Marco Pantani, legend of cycling, during an ascent of the *Giro d'Italia, 1999.* (From http://www.strettoweb.com/)

choice often embraces difficulties regarding lots of tasks. For instance, suppose that we want to build an architecture able to recognize the presence of a bicycle in a photograph (see Fig. 1.1).
We know that a bicycle has wheels, bike pedals, handlebars, so we might check the presence of these features in the image. Unfortunately it is difficult to formalize what they are in terms of pixels: for instance their simple geometric shape may be partly obscured by other elements or by shadows falling on it. Machine Learning can help us to discover not only the mapping function from input to output, but also the representation itself. This approach is known as *representation learning*: it improves performances of the algorithm and allows humans to save time and effort. Indeed, a good set of features can be discovered in few minutes (simple task), days or months (complex tasks) which is far away from the time needed to a community of researchers.
An emblematic example of representation learning is the *autoencoder*. It is composed by two parts, an encoder and a decoder. Combining them, an autoencoder tries to reconstruct an input from an intermediate representation (given by an hidden layer). In other words it attempts to approximate the identity function. Different kinds of autoencoders can be used to achieve diverse targets. Generally, the common factor between them is that they allow us to maintain the relevant information of the features (sometimes unobservable directly), founding a more efficient representation.

However, when analysing an object, such as Fig. 1.1 , different factors of variation may occur. Some of them might be the angle of the sun, the colours, the position of the bicycle. In addition the shadows in the image could make the pink pixels very close to the black ones. Therefore, extracting features from a photograph can be very difficult. When obtaining a representation is as difficult as to solve the original problem, representation learning does not seem to help us at first glance.
Fortunately **deep learning** overcomes the weak points of representation learning building complex representations in terms of simpler ones. Indeed, this system combines simpler concepts, built according to a hierarchy, to represent an image; for example considering the contours which are defined theirself in terms of edges. The emblematic instance of a deep learning model is the *multilayer perceptron* (MLP) (see Fig. 1.2). It represents a mathematical function that maps an input to an output. This function is composed by simpler functions, defined by the whole layers.
Besides, deep learning introduces a key concept: the *depth*. Networks with great depth can execute several instructions together offering also an enormous power since later instructions can refer back to the result of the earlier ones.

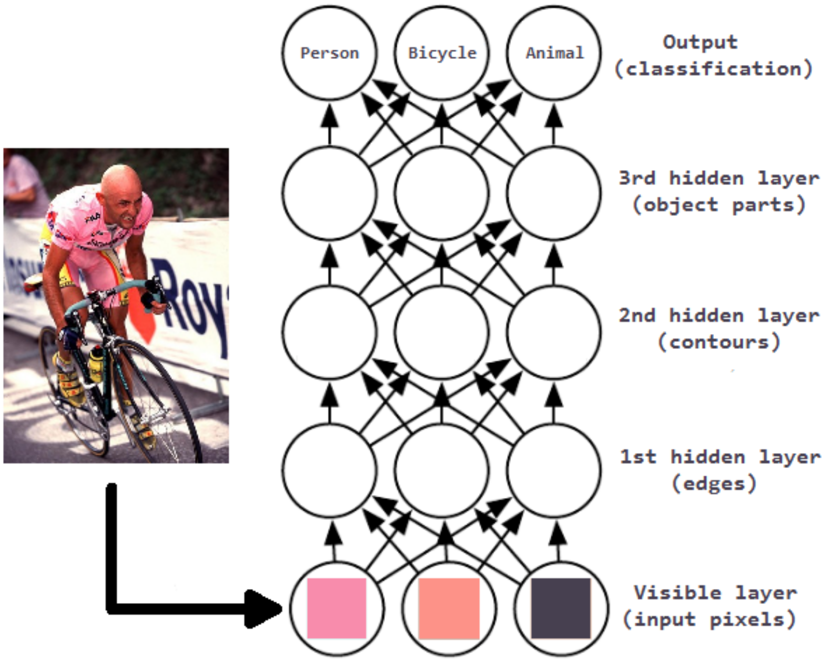


Figure 1.2: Example of MLP applied to Fig. 1.1 . It seems an insurmountable task to find a mapping from raw input data (set of pixels) to an object identity. **Deep learning** overcomes this obstacle by breaking the desired complicated mapping into a series of nested simple functions, each represented by a different layer of the model. The input is presented to the visible layer, so called because it contains the variables that we are able to observe. Successively a series of "hidden" layers, so named because their values are not given in the data, abstract features from the photograph. Thus, given the pixels, the first hidden layer identifies edges by comparing the brightness of adjacent pixels. Then the second hidden layer searches for contours, identifiable as a collection of edges. The third hidden layer determines the entire parts of the object by finding a specific collection of contours. Finally these parts are used to classify the object in the image as a single element.

1.1 History of deep learning

Even though deep learning is widely known as an emerging and exciting field, its roots date back to the far 1940s. Deep learning only seems to be a new technology because it has been relatively unpopular for several years and characterized by several names.
Largely speaking, it is possible to identify three main epochs related to its development:

- *1940s-1960s, cybernetics*;
- *1980s-1990s, connectionism*;
- *from 2006 onwards, deep learning.*

The earliest learning algorithms were intended to be computational models of biological learning, i.e. mainly referred to how learning process occurs in the brain. Consequently, deep learning was also known as *artificial neural networks* (ANNs). However, ANNs were not designed to be realistic models of biological functions. Indeed deep learning goes beyond the neuroscientific perspective of the ANNs; it is a more general tool not necessarily neurally inspired belonging to the machine learning field.
The earliest predecessors of modern deep learning were the so called *linear classifiers*. Essentially they receive an input vector $x = (x_1, ..., x_n)$ and, learning a set of weights $w = (w_1, ..., w_n)$, compute an output $y = f(x, w) = x_1 w_1 + .. + x_n w_n$. The early model of brain function was the *McCulloch-Pitts Neuron* (1943). It was able to correctly classify two categories of inputs, by evaluating if $f(x, w)$ is positive or negative. In this model the weights needed to be correctly set by an external supervisor. In the 1950s, the *perceptron* (Rosenblatt, 1958, 1962) became the first model that could learn weights defining categories using examples of inputs taken from each category. The *adaptive linear element* (ADALINE), which dates from about the same time, simply returned the value of $f(x)$ itself to predict a real number, and could also learn to forecast these numbers from data. However, linear models have many limitations. The most famous regards their inability to learn the **XOR** function, where $f([0, 1], w) = 1$, $f([1, 0], w) = 1$ but $f([1, 1], w) = 0$ and $f([0, 0], w) = 0$. Critics towards these limitations caused a backlash against biologically inspired learning. In spite of this, neuroscience has become an important source of inspiration for deep learning researchers, even if it is no longer the predominant guide for the field. The main reason is that we simply do not have enough information about the brain to use it as a guide. To obtain them, we would need to be able to monitor the activity of (at the very least) thousands of interconnected neurons simultaneously. Because it is not yet possible, we are far from a deep understanding of the brain.
These simple learning algorithms greatly influenced the modern scenario of machine learning. For example, the training algorithm used to adapt the weights of the ADALINE was a special case of an algorithm called *stochastic gradient descent*. Moreover, slightly modified versions of the stochastic gradient descent algorithm remain the dominant training algorithms for deep learning models.

In the 1980s, the second wave of neural network researches emerged in great part trough a movement called *connectionism* or *parallel distributed processing*. Connectionism arose in the context of cognitive science. The connectionists began to study models of cognition that could actually be grounded in neural implementations, reviving many ideas dating back to the work of psychologist Donald Hebb in the 1940s. The central idea in connectionism is that a large number of simple computational units can achieve intelligent behaviour when networked together. An emblematic example is the *Hopfield network*, a form of recurrent artificial neural network popularized by John Hopfield in 1982.
This network is characterized by binary threshold units, i.e. the units only take on two different values for their states and their value is determined by whether or not the input of the units exceeds their threshold. Conventionally, the value of these units is 1 or -1. However, it is possible to find literature using 0 and 1 as well.
Every pair of units i and j in a Hopfield network have a connection that is described by the connectivity weight w_{ij}. Therefore the Hopfield network can be formally described as a complete undirected graph $G = (V, f)$, where V is a set of McCulloch-Pitts neurons and $f : V^2 \to \Re$ is a function that links pairs of nodes to a real value, the connectivity weight.
The links in a Hopfield network typically have the following restrictions:

- $w_{ii} = 0, \forall i$: no unit has a connection with itself;
- $w_{ij} = w_{ji}, \forall i, j$: the connections are symmetric. This constraint guarantees the *energy* function decreases monotonically.

Indeed, Hopfield networks have a scalar value associated with each state of the network known as the energy E of the network, where:

$$E = -\frac{1}{2} \sum_{i,j} (w_{ij} s_i s_j) + \sum_i (\theta_i s_i) \tag{1.1}$$

This value is called the energy since the definition ensures that when units are randomly chosen to update, the energy E will either lower in value or stay the same. Furthermore, under repeated updating the network will eventually converge to a state which is a local minimum in the energy function (which is considered to be a *Lyapunov function*). Thus, if a state is a local minimum in the energy function, it is a stable state for the network.
Furthermore, Hopfield model accounts for *associative memory* through the incorporation of memory vectors. This capability arises from the collective properties of Hopfield model and it is very useful for recognition of corrupted inputs and missing information recovery. An instance of associative memory might be our capability of images recognition when they are corrupted.
Several key concepts arose during the connectionism movement of the 1980s and still have a central role in deep learning. One of these is the *distributed representation*. To understand it, consider for instance to have a vision system that can recognize cars, trucks, and birds and these objects can each be red, green, or blue. One way of representing these inputs could be to have a separate neuron or hidden unit activating for each of the nine possible combinations: red truck, red car, red

bird, green truck, and so on. This requires nine different neurons, and each neuron must independently learn the concept of colour and object identity. One way to improve on this situation is to use a distributed representation, with three neurons describing the color and three neurons describing the object identity. This requires only six neurons instead of nine, and the neuron describing redness is able to learn about redness from images of cars, trucks and birds, not only from images of one specific category of objects. Other two major accomplishments of the connectionist movement were the successful use of *back-propagation* to train deep neural networks with internal representations and the popularization of the back-propagation algorithm, which is still the dominant approach to train deep models.
During the 1990s, researchers made important steps forward in modelling sequences with neural networks. Hochreiter (1991) and Bengio et al. (1994) identified some of the fundamental mathematical difficulties in modelling long sequences. Hochreiter and Schmidhuber (1997) introduced the *long short-term memory* or LSTM network to solve some of these difficulties. LSTM is widely used for many sequence modelling tasks, including many natural language processing tasks at Google. The second wave of neural networks research lasted until the mid 1990s. Again, there was a decline in deep learning research, due to unrealistic ambitious tasks that the existing systems were supposed to solve. At this point in time, deep networks were generally believed to be very difficult to train. Instead those algorithms, existing since the 1980s, work quite well, but this was not ascertained circa 2006. The issue is perhaps simply that these algorithms were too computationally costly to allow experimentation with the hardware available at the time.
The advent of more powerful computers has permitted the renaissance of deep learning. The third wave began with a focus on new unsupervised learning techniques and the ability of deep models to generalize well from small datasets. Over time this tendency has changed and researchers has become more interested in much older supervised learning algorithms and in the ability of deep models to leverage large labelled datasets.

1.2 What did make deep learning such a crucial technology?

As seen before, the origin of deep learning dates back to 1940s. In addition, most of the techniques used are nearly identical to the learning algorithm that struggled to solve simple problems in the 1980s, even though the models we train with this algorithm have simplified training procedures. So, what has allowed deep learning to become such a crucial technology?
Two key reasons compose the answer to this question.

1.2.1 Dataset size

The achievement of good performances from a deep learning algorithm requires a sufficient amount of skills. Fortunately, by increasing dataset sizes it is possible to reduce the necessary amount of skill. Therefore, we can establish and provide the algorithms with the resources they need to succeed. Besides, more and more of human activities take place on computers, thus also the amount of recorded information about what we perform grows. Since our computers are increasingly linked together, it becomes easier to collect these records into a dataset appropriate for machine learning applications. This new resource allows the supervised learning algorithm to achieve acceptable results with around 5000 labelled features and to overcome human performance when trained with 10 billions of labelled examples. In few words, the age of "Big Data" has made machine learning much easier.

1.2.2 Model size

The second key reason is that we have become able to build larger models, thanks to the computational development achieved. One of the fundamental ideas of the connectionism was that intelligence raises when many neurons work together. Therefore a small set of units is not useful.
However it is known that biological systems are not densely connected. The key role is played by the total number of neurons that was astonishingly small until recent years. Since the introduction of hidden layers, ANNs have doubled in size roughly every 2.4 years. This growth has been driven by faster computers with larger memory and by the availability of larger datasets. Larger networks are able to achieve higher accuracy on more complex tasks. This trend seems to continue for decades. Unless new technologies allow faster scaling, ANNs will not have the same number of neurons as the human brain until at least the 2050s. Furthermore the results achieved, computationally speaking, are not enough to reach the number of neurons of a vertebrate animal like a frog. However, the increase in model size over time, due to the availability of faster CPUs, the advent of general purpose GPUs, faster network connectivity and better software infrastructure for distributed computing, is one of the most important trends in the history of deep learning. This trend is generally expected to continue well into the future.

1.3 Learning algorithm

As aforesaid, a machine learning algorithm is able to learn from data. In 1997, Mitchell provided definition of what "learning" means:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Therefore it is a well worth to focus on T, E and P.

1.3.1 The task T

Machine learning not only allows us to tackle tasks that are too complicated to solve with programs written and designed by human beings, but its understanding develops our knowledge about the principles that lies behind intelligence. With

task we refer to the target which the system, via learning from examples, has to achieve. An example is a collection of features that we want the machine to process. Mathematically speaking, it is a vector $x \in \Re^n$ whose each entry is a feature.
Some of the most common tasks machine learning allows to solve are:

- *Classification*: in this case, the computer program is asked to specify which of k categories an input belongs to. To solve this task, the learning algorithm searches for a function $f(x) : \Re^n \mapsto \{1, ..., k\}$.
  An example of this kind of activity is the aforementioned *object recognition*, the same basic technology that allows computers to recognize faces, which can be used to automatically tag people in photo collections and that permits users to interact more naturally with their computer.
  Classification becomes more challenging when some inputs are missing. In this case, the algorithm must learn a set of mapping $f_i(x)$, each of them representing a classification of x with a different set of missing inputs. This situation is typical of medical diagnosis, because many kinds of medical tests are expensive or invasive. One way to efficiently define such a large set of functions is to learn a probability distribution over all the relevant variables, then solve the classification task by marginalizing out the missing variables.

- *Regression*: in this case, the computer program is asked to predict a numerical value given some inputs. To solve this task, the learning algorithm looks for a function $f(x) : \Re^n \mapsto \Re$. For instance the prediction of future stock prices is a regression task.

- *Transcription*: in this case, the computer program is asked to observe a relatively unstructured representation of some kind of data and transcribe it into discrete, textual form. For example Google Street View uses deep learning to solve a transcription task: to process address numbers from a photograph containing them. The machine learning algorithm is asked to return the numbers.

- *Translation*: in this kind of task, the input consists in a sequence of symbols, expressed in a specific language, and the algorithm is requested to translate them in other symbolic sequence with a different language.

- *Structured output*: this class involves any task whose output is a vector (or other structured data containing multiple values) characterized by a strong relationships among the different elements composing it. This is a broad category that includes the transcription and translation tasks described above, but also many other ones. An example is the parsing—mapping of a natural language sentence into a tree that describes its grammatical structure whose tagging nodes are verbs, nouns, or adverbs, and so on.

- *Anomaly detection*: during this type of task, the computer program sifts through a set of events or objects, and flags some of them as being unusual or atypical. An example of an anomaly detection task is credit card fraud detection. By modelling your purchasing habits, a credit card company can detect misuse of your cards. Thus, if a thief steals your credit card or the information in it, the purchases of the thief will often come from a different probability distribution compared to your own one. The credit card company can prevent fraud by placing an hold on an account as soon as that card has been used for an uncharacteristic purchase.

There are many other tasks that practitioners attempt to solve, such as *synthesis and sampling*, *imputation of missing values*, *denoising* and *probability mass function estimation*.

1.3.2 The performance P

The performance P of a machine learning algorithm is a measure required to evaluate its abilities. For classification task, classification with missing input task and transcription usually *accuracy* is computed. This quantity represents the portion of correct outputs out of the total number of examples proposed. Obviously, it is possible to define the accuracy in terms of the proportion of examples for which the model produce incorrect output. It is a common practice to consider the error rate as the expected 0-1 loss, which is 0 if the example is correctly classified, 1 otherwise. For other kind of tasks, the most common approach is to report the *average log-probability* assigned by the model to some examples. Furthermore, the performances are evaluated on a *test set* of examples, different from the one used for training the algorithm (*training set*).
The choice of performance measure is of key importance but often it is hard. This is because in some cases we do not know well what should be the scale of such a measure or we know what we want to measure but it is not possible in practice to obtain that quantity. For instance, consider a translation task: Are we interested in the accuracy of the system at transcribing sentences or rether in a more fine-grained performance?

1.3.3 The experience E

Last, but not least, we find the experience. Mainly, machine learning algorithms are divided in two classes according to the kind of experience they are allowed to have during the learning process: *supervised* and *unsupervised* algorithms. Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target. The term supervised learning originates from the view of the target y being provided by an instructor or teacher who shows the machine learning system what to do. In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide. Roughly speaking, unsupervised learning involves observing several instances of a random vector x and attempting to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution. Instead supervised learning involves observing several examples of a random vector x and an associated value or vector y, and learning to predict y from x, usually by estimating $p(y|x)$. However the line between supervised and unsupervised learning is not

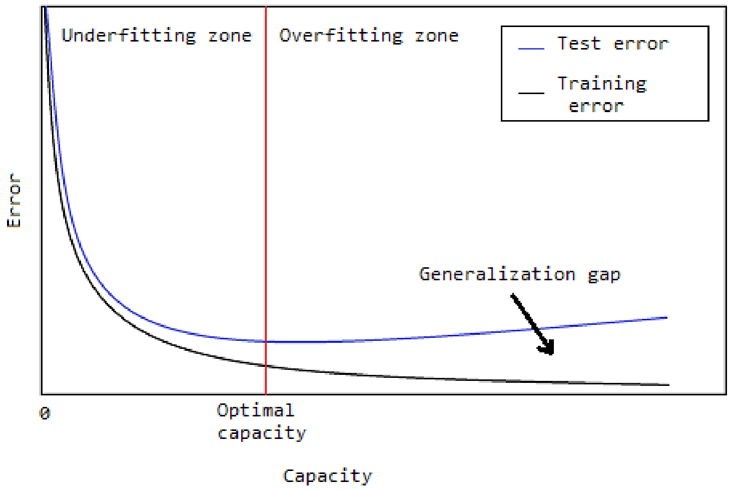


Figure 1.3: Relationship between capacity and error. After a first phase where both train and test error are in an high (*underfitting regime*), the capacity increses reaching the *optimal capacity*. Therefore the capacity beacomes too large and we enter in the *overfitting regime*.

so marked, indeed often machine learning technologies are used to challenge both tasks.

1.4 Underfitting and overfitting

The central challenge in machine learning is the *generalization*: not only to perform well on our training set but even on previously unseen inputs. When we train an algorithm we want to cut down the *training error* by minimizing the loss (cost) function. Therefore, so far we have dealt with an optimization problem. What makes machine learning different from optimization is that we want more: to make as small as possible the generalization error, known as *test error*. This one is defined as the expected value of the error on a new input. The goodness of a machine learning algorithm performance is determined by its ability to:

1. make the training error small;

2. make the gap between the training and test error small.

These two factors correspond to the two central challenges in machine learning: *underfitting* and *overfitting*. The former occurs when the model is not able to obtain a sufficiently low error value on the training set. The latter occurs when the gap between the training error and test error is too large (see Fig. 1.3).
We can control whether a model is more likely to underfit or overfit by altering its *capacity*. The capacity of a model is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high

capacity can overfit by memorizing properties of the training set that do not serve them to perform well on the test set. One way to check the capacity of a learning algorithm is the specification of the solution as its hypothesis space, the set of functions that the learning algorithm is allowed to select. For instance, the hypothesis space of a linear regression algorithm is the set of all linear functions of its input. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. By doing so we increase the capacity of the model.

1.5 The *No Free Lunch Theorem* and regularization

Learning theory claims that a machine learning algorithm can generalize well from a finite training set of examples. This assert is formalized in *The No Free Lunch Theorem* for machine learning (Wolpert, 1996). It states that, if averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. In other words, in some sense, no machine learning algorithm is universally better than any other. The most sophisticated algorithm we can conceive has the same average performance (over all possible tasks) as merely predicting that every point belongs to the same class.
It is well worth to remark that this result holds only **averaging** on all possible data generating distributions. Making assumptions on the dataset we have in front of us, it is possible to achieve good performances. Consequently the target of machine learning is not to generate an *universal algorithm*, instead it is to understand what kind of distributions are relevant to *artificial intelligence* (AI) to face the real world.

To improve learning performances we can act in various ways: working on the capacity, restricting the solution set introducing some hypothesis or even expressing a preference for one type of solution. There are many methods to define such preferences, together they are know as *regularization*: it is any modification we make to a learning algorithm that is intended to reduce its test error but not its training error. One of the most common approaches of this type is the *weight decay*. This procedure allows to express we prefer those weights with a smaller squared L^2 norm.
The No Free Lunch Theorem has made it clear that there is no best machine learning algorithm, and, in particular, no best form of regularization. Instead we must choose a form of regularization that is well-suited to the particular task we want to solve.

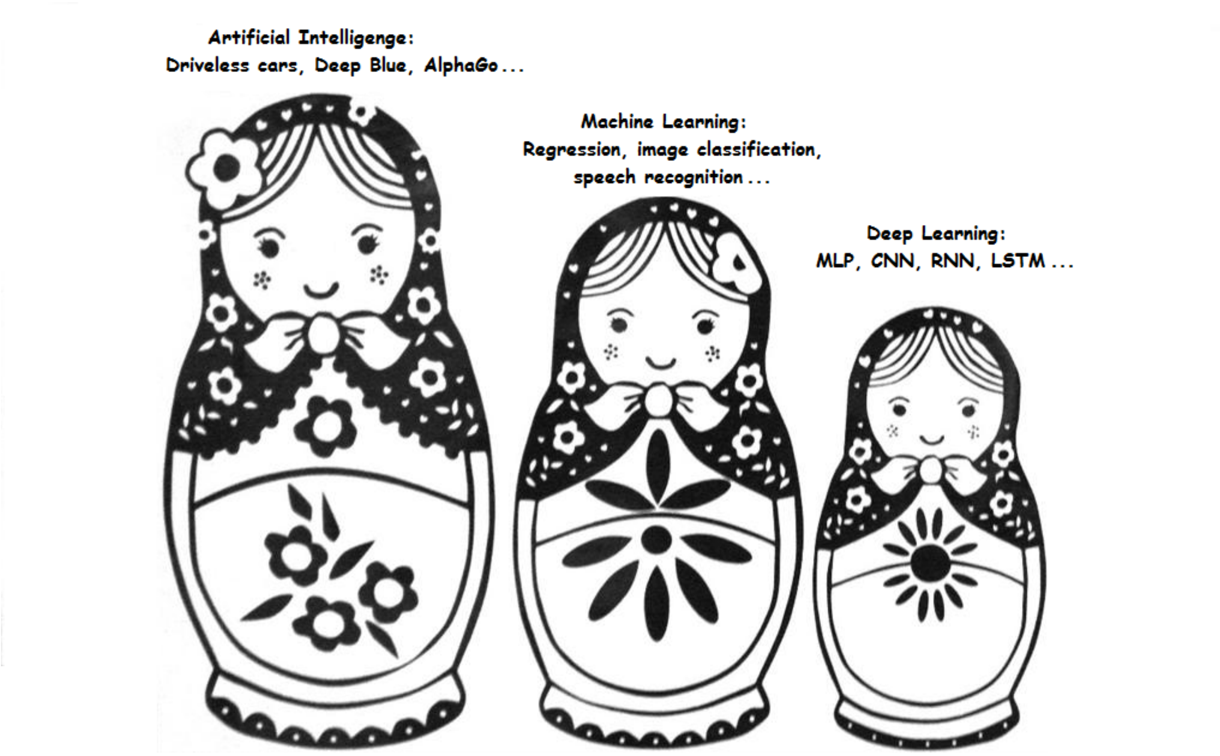


Figure 1.4: It is possible to think of artificial intelligence, machine learning and deep learning as a set of Russian dolls nested within each other. Deep learning is a subset of machine learning, which is a subset of AI. Roughly speaking AI is human intelligence exhibited by machines; machine learning is an approach to achieve AI; deep learning a technique for implementing machine learning. (From www.pinterest.com)

Chapter 2

TensorFlow

Experimentation in the field of deep learning is made possible by the existence of diverse libraries concerning machine learning: Theano, Caffe, Torch. Between these tools, we find **TensorFlow**(TF), the software which we intend to use.
TF was developed by the *Google Brain team* open-sourced by Google in late 2015. Previously it was exclusively used by Google in speech recognition, Search, Photos and Gmail, in addition to many others applications.
Moreover this library can be implemented in C++ and Python. Furthermore it is very flexible and it is possible to exploit the calculation power of more than one CPU or GPU on desktop, server or mobile devices with a unique API. TF facilitates deep learning since the differentiation is made automatic by several routines: therefore the implementation of gradient based algorithms becomes quite easy. In addition, several operations are pre-compiled. This aspect accelerates the creation of machine learning prototypes.
TF developers curated the visualization aspect too. Indeed a characteristic feature of this library is its interactive visualization environment called TensorBoard. This tool shows a flowchart of how data transform, displays summary logs over time, as well as traces performances.
This chapter introduces the fundamental aspects of TF, showing some instances of simple codes too.

Figure 2.1: TensorFlow logo.

2.1 Tensor

It is widely known that Python is a succinct language. If, on one hand, this implies both brevity of code and easy comprehension, on the other hand it means a lot is happening behind each line of code. Machine learning algorithm requires a large amount of mathematical operations, in many cases it relies on the composition of simple functions iterated until convergence. To perform these computations it is possible to use standard programming language for sure, but to have both manageable and efficient code the secret is to exploit a well-written library, such as TF. By installing TF, we also obtain a well known Python library named Numpy, which eases mathematical operations in Python.
As aforesaid in the previous chapter, a common way to represent an object is to describe it in terms of features, collected in *feature vector* which in turn composed a *matrix*. The syntax to represent matrices in TF is a vector of vectors, each of the same length (see Fig.2.2).

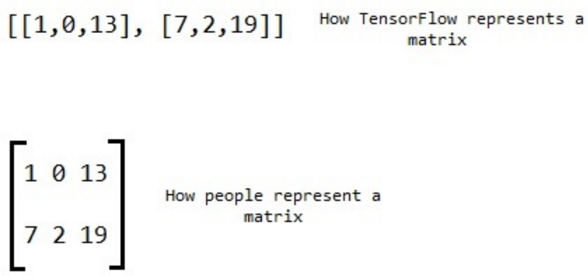


Figure 2.2: A matrix in TensorFlow.

Every element is identified by row and column indices, so we can access to it by specifying them. Sometimes it is more convenient to use more than two indices, therefore we define a generalization of a matrix, a *tensor*. The syntax of a tensor is very nested. For instance a 2-by-2-by-3 tensor is composed by two matrices, each of size 2-by-3 (see Fig.2.3).

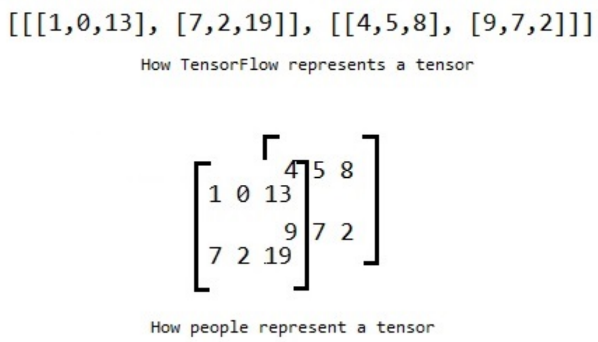


Figure 2.3: A tensor in TensorFlow.

In the following few lines of code is shown how to generate a tensor in TF.
As showed by the output, the tensor is represented by the aptly name Tensor object. Each tensor in TF has a unique label (`name`), a dimension(`shape`, 1-dimensional

```
import tensorflow as tf

myTensor = tf.constant([[[1,0,13],
             [7,2,19]],
            [[4,5,8],
             [9,7,2]]])

print(myTensor)
```

```
Python 3.5.2+ (default, Sep 22 2016, 12:18:14)
[GCC 6.2.0 20160927] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
============ RESTART: /home/gabriele/MyTensorFlowStuff/Thesys.py ===
=========
Tensor("Const:0", shape=(2, 2, 3), dtype=int32)
```

Figure 2.4

tensor of `int32`, i.e. a list of integers) that represents its structure and data type (`dtype`) which specifies the kind of values it includes. Because the name was not provided, the library assigns automatically the label `"Const:0"`. Furthermore TF has some convenient command to create simple tensors. For instance `tf.zeros(shape)` generates a tensor initialized at zero, `tf.ones(shape)` creates a tensor which has all entries initialized to one.

2.2 Operators

Once that we have a tensor, it is possible to apply interesting operators to build mathematical function. In addition to those relative to the most common operations such as addition or multiplication, we find the *negation operators*. This one takes a tensor as input and returns the same tensor but with all entries negated. Some of the most used TF operators are reported below:

- `tf.add(x, y, name=None)`: Adds two tensors of the same type;
- `tf.sub(x, y, name=None)`: Subtracts two tensors of the same type;
- `tf.multiply(x, y, name=None)`: Multiplies two tensors element-wise;
- `tf.div(x, y, name=None)`: returns the element-wise division of two tensors;
- `tf.truediv(x, y, name=None)`:divides two tensors returning floating point results;
- `tf.floordiv(x, y, name=None)`: divides to tensors element-wise, rounding toward the most negative integer;
- `tf.mod(x, y, name=None)`:returns element-wise remainder of division;
- `tf.pow(x, y, name=None)`: returns the element-wise power of x to y;
- `tf.sqrt(x, name=None)`: computes square root of x element-wise;

- `tf.exp(x, name=None)`: computes exponential of x element-wise.

It is possible to find the official documentation at:
`https://www.tensorflow.org/versions/r0.12/api_docs/python/math_ops.html`.
Furthermore, TF algorithms are pretty easy to visualize. They can be described by flowcharts. Indeed, it is possible to think to every operator as a node of a graph. Moreover, consider the edges in the graph represent composition of the mathematical functions. Therefore, supposing that we want to visualize the *negation operator*, it is nothing else that a node where its incoming and outgoing edges describe how the tensor transforms.

2.3 Session

A *session* in TF is what permits to execute our code. In particular, it specifies how the lines of machine learning code should run, therefore we can later configure the session to change its behaviour without changing the code.
What we need to do, is to create a session class using `tf.Session()` and specify it to run an operator written in the code.

```
import tensorflow as tf

myTensor = tf.constant([[[1,0,13],
                         [7,2,19]],
                        [[4,5,8],
                         [9,7,2]]])

neg_myTensor = tf.neg(myTensor)

with tf.Session() as sess:
    Neg_myTensor = sess.run(neg_myTensor)

print(Neg_myTensor)
```

```
Python 3.5.2+ (default, Sep 22 2016, 12:18:14)
[GCC 6.2.0 20160927] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: /home/gabriele/MyTensorFlowStuff/Thesys.py ===========
[[[ -1   0 -13]
  [ -7  -2 -19]]

 [[ -4  -5  -8]
  [ -9  -7  -2]]]
```

Figure 2.5: Example of session. In this case, we simply negate the previously defined tensor.

A session may own resources, such as *variables* or *queues*. It is important to release them when they are no longer required. To do this, we can invoke the *close* method on the session or use the session as a context manager (as shown in Fig. 2.5).
Moreover, it is also possible to pass options to `tf.Session`. The `ConfigProto` protocol exposes various configuration options for a session. For instance, to create a session that automatically determines the best way to assign a CPU or a GPU device to an operation and log the resulting placement decisions, run the code in Fig. 2.6 .

```python
import tensorflow as tf

myTensor = tf.constant([[[1,0,13],
                         [7,2,19]],
                        [[4,5,8],
                         [9,7,2]]])

neg_myTensor = tf.neg(myTensor)

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    Neg_myTensor = sess.run(neg_myTensor)

print(Neg_myTensor)
```

Figure 2.6: Example of how log a session.

The inputs of a session can be three. So far we have managed constants, but it is possible to use *placeholders* and *variables*. In particular:

- `tf.placeholder(dtype, shape=None, name= None)` inserts a placeholder for a tensor that will always have to be fed. This tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to `Session.run()`,`Tensor.eval()` or `Operation.run()`;

- `tf.Variable()` creates an instance of the python class `Variable`. It requires an initial value for the variable, which can be a tensor of any type and shape. After construction, both type and shape of the variable are fixed, what we can change by using assign methods is its value. Variables in our code might be parameters of a machine learning model.

- `tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)` creates a constant tensor. The resulting tensor is populated with values of type `dtype`, as specified by the argument `value` and optionally `shape`.

Sometimes it is necessary to use TF in an interactive environment (for instance when we have to make a presentation). In such a situation it is easier to create an interactive session, where the session is implicitly part of any call to `eval()`. This function belong to every tensor in TF, and when we call it, is equivalent to use `sess.run(...)`. To exercise the interactive nature we can exploit *Jupyter*. It is a web application that shows computation elegantly so that we can share our code with others. Therefore via Jupyter, we can share ideas and learn about other uploaded codes.
Everything in Jupyter notebook is an independent chunk of code called cell. This helps us to divide in manageable and more comprehensive parts our original code. Each cell can be run individually or we can choose to evaluate the whole parts at once as well. There are mainly three ways to run cells:

- `Shift+Enter`: evaluates the cell and passes to the next cell below;

- `Ctrl+Enter`: executes the cell and maintains the cursor on it;

- `Alt+Enter`: runs a cell and then inserts a new empty cell below.

Moreover it is possible to change the cell type by the toolbar or alternatively by pressing `Esc` to leave the edit mode, use the arrow to highlight the cell we are interested in, then press `Y` to change it to `code` or `M` for markdown mode (see Fig. 2.7).

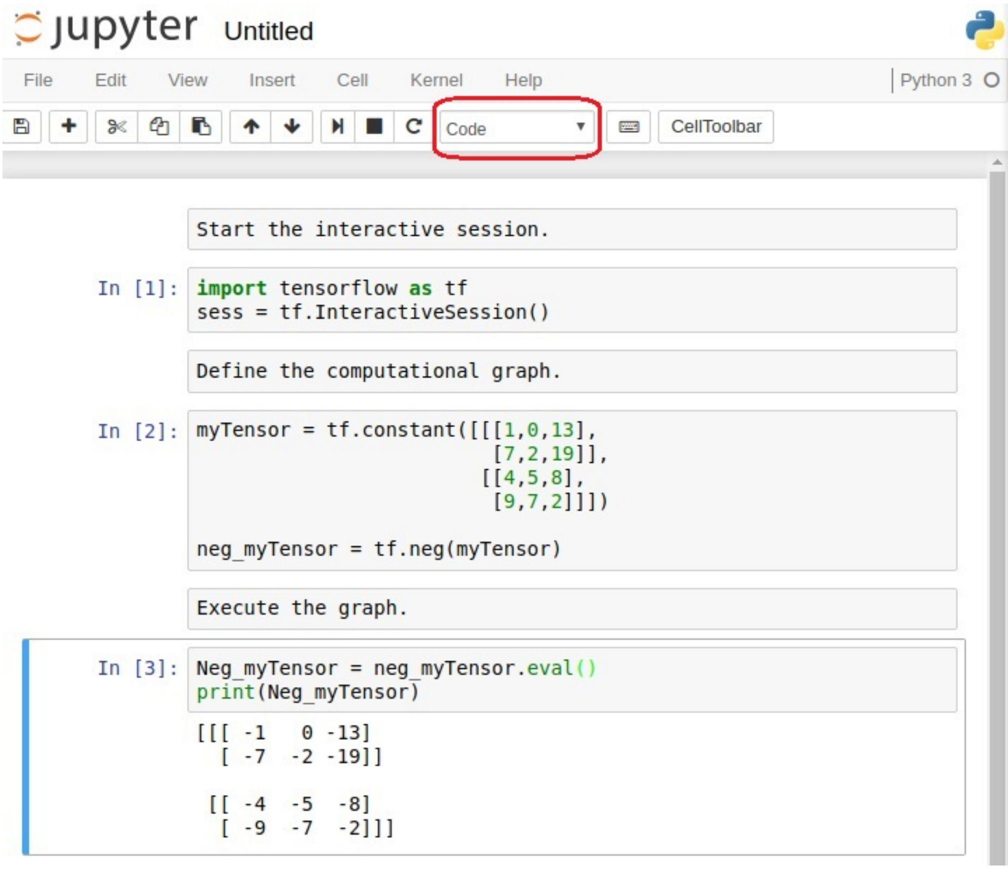


Figure 2.7: Example of Jupiter notebook. There are shown 6 cells, three of text and three of code respectively. The red box highlights the dropdown menù where we can change the cell type.

2.4 Variables: initialization, saving and loading

When we train a model, we use *variables* to hold and update parameters. They must be explicitly initialized and can be saved to disk, during and after training. Furthermore, we can load the saved values to analyse the model.
Variable initializers must be run explicitly at the beginning of the train phase. In order to run variables we can either add an operation that runs all the variable initializers or explicitly run the command `tf.initialize_all_variables()`. In both cases we must run the operations after we have fully built the model and launched it in a session. The convenience function `tf.initialize_all_variables()` initializes all variables in the model. In Fig. 2.8 is shown an example of how to use variables. Suppose we are interested in the computation of the average through time of a particular quantity. Well, we need just to add up all the values and divide by their

total number. If the total number is unknown we can apply the *exponential averaging* technique which estimates the current average as a function of the previous estimated average and the current value. Mathematically speaking:

$$Avg_t = f(Avg_{t-1}, x_t) = (1 - \alpha)Avg_{t-1} + \alpha x_t \tag{2.1}$$

where α is a parameter that describes how strongly recent values should be biased in the calculation of the average (the higher is α, the most the estimated will differ from the previous one).

```
import tensorflow as tf
import numpy as np

#creates a vector of 100  numbers with a mean
#of 10 and a variance of 1
raw_data = np.random.normal(10, 1, 100)

alpha = tf.constant(0.05)
curr_value = tf.placeholder(tf.float32) #this value will be
                                        #injected from the session
prev_avg = tf.Variable(0.)
update_avg = alpha * curr_value + (1 - alpha) * prev_avg


with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    for i in range(len(raw_data)):
        curr_avg = sess.run(update_avg, feed_dict={curr_value:raw_data[i]})
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)
```

Figure 2.8: Example of how to use a variable. A placeholder is like a variable but its value is injected from the session.

As aforementioned we can save and restore a model as well. The easiest way to do it is using the `tf.train.Saver` object. The saver object provides methods to `save` and `restore` operations. To run them, we need to specify paths for the *checkpoint files*. The checkpoint files are binary files which contain a map from the variable names to the tensor values. When we create a `Saver` object, we can optionally choose names for the variables in the checkpoint file. The default names are the `Variable.name` property for each variable. Fig. 2.9 displays an example.
It is possible to notice an `average.ckpt` file in the same directory as the source code. It is the checkpoint file, that we can use to retrieve the data using the `restore` function.

```
import tensorflow as tf
import numpy as np

#creates a vector of 100  numbers with a mean
#of 10 and a variance of 1
raw_data = np.random.normal(10, 1, 100)

alpha = tf.constant(0.05)
curr_value = tf.placeholder(tf.float32) #this value will be
                                        #injected from the session
prev_avg = tf.Variable(0.)
update_avg = alpha * curr_value + (1 - alpha) * prev_avg


with tf.Session() as sess:
    saver = tf.train.Saver() #saver operator
    sess.run(tf.initialize_all_variables())

    for i in range(len(raw_data)):
        curr_avg = sess.run(update_avg, feed_dict={curr_value:raw_data[i]})
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)

    save_path = saver.save(sess, "average.ckpt")
    print("Averages data saved in file: %s" % save_path)
```

Figure 2.9: Example of how to use the `Saver` object. The saver operation enables saving and restoring variables. If no list is passed into the constructor, then it saves all variables in the current program.

2.5 TensorBoard

In order to easy the understanding, debug and the optimize the computations we use in TF, it is possible to exploit a visualization tool, named *TensorBoard* (TB). We can use it to visualize our TF graph, plot quantitative metrics about the execution of the graph and show additional data like images which pass trough it.
This tool operates reading the summary data that we generate running TF. Therefore, first of all we have to create TF graph that we would like to collect summary data from, and decide which nodes we would like to annotate with summary operations. In particular, we can handle these by attaching `scalar_summary` operations to them, giving meaningful tags as well.
In order to generate summaries, we need to run all of these summary nodes. Managing them by hand would be tedious, so use `tf.summary.merge_all` to combine them into a single operation that generates all the summary data. At this point, we can run the merged summary operation, which will generate a serialized `Summary protocol buffer` (protobuf) object with all of our summary data at a given step. Finally, in order to write this summary data to disk, we pass the summary protobuf to a `tf.train.SummaryWriter`.
The `SummaryWriter` takes a log directory in its constructor. It is important since it represents the directory where all the events will be written out. Besides, the `SummaryWriter` can optionally take a graph in its constructor. If it receives a graph object, therefore TB will visualize our graph along with tensor shape information. This will give us a much better sense of what flows through the graph. In Fig. 2.11 is displayed a very simple example.
Before starting TB, we need to create a directory called `logs` in the same folder as our source code. At this point, we can run TB by the following command:
`$ tensorboard -logdir=./logs`

Once TB is running, we can navigate our web browser to `localhost:6006` to view the TB (see Fig. 2.12).

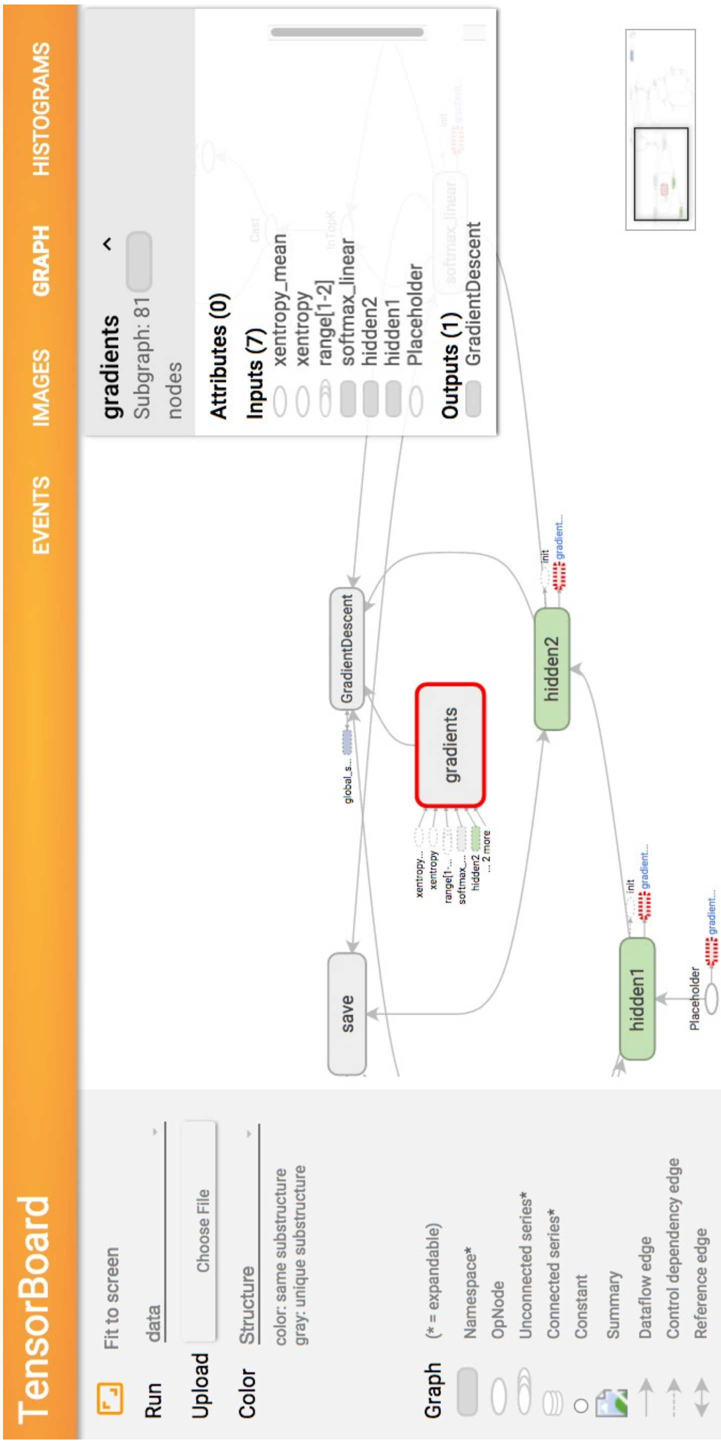


Figure 2.10: The figure shows a detail of TB computational graph. More precisely the arrows represent the flux of the tensor while the nodes describe the computation applied to it. As we can notice, it is possible to inspect each node. This is particularly useful in order to debug our model and understand better what is happening.
For instance consider the node `gradients`: this variable has as scope the application of the gradient descent to 7 inputs, propagating it from the first to the last. (From *Christopher Bourez's blog*)

```
import tensorflow as tf
import numpy as np

#creates a vector of 100  numbers with a mean
#of 10 and a variance of 1
raw_data = np.random.normal(10, 1, 100)

alpha = tf.constant(0.05)
curr_value = tf.placeholder(tf.float32) #this value will be
                                        #injected from the session
prev_avg = tf.Variable(0.)
update_avg = alpha * curr_value + (1 - alpha) * prev_avg

#create summury nodes for the averages and the values
avg_hist = tf.scalar_summary("running avarage", update_avg)
value_hist = tf.scalar_summary("incoming value", curr_value)

#merge the summury and pass the logdir to the writer
merged = tf.merge_all_summaries()
writer = tf.train.SummaryWriter("./logs")

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())

    for i in range(len(raw_data)):
        #run all the operations at the same time
        summary_str, curr_avg = sess.run([merged, update_avg], feed_dict={curr_value:raw_data[i]})
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)
        #add the summary to the writer
        writer.add_summary(summary_str, i)
```

Figure 2.11: Example of how to create summary nodes and run these operations.

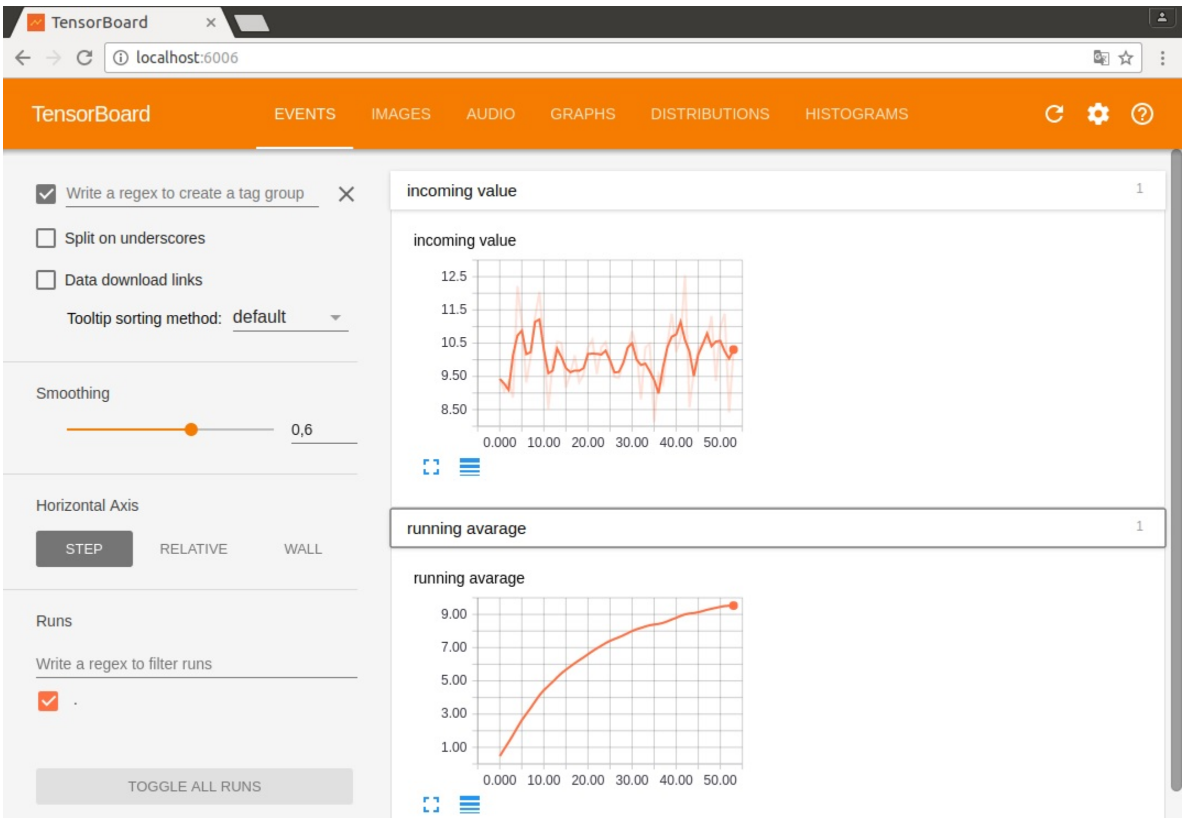


Figure 2.12: Statistics relative to our source code, showed by TB.

Chapter 3

Applications to MNIST dataset

In this chapter we are going to show some examples of machine learning algorithms applied to MNIST dataset. They are re-elaborated versions of some tutorials proposed on `www.tensorflow.org`.
Even though these models are classical introductive basic instances, they are very useful to approach some techniques applied throughout the next two chapters. In particular, the following examples introduce the reader to our programming style. Furthermore, they highlight the importance of depth and how much versatile deep learning algorithms are.
More precisely the models we will consider are *Multilayer Perceptron*, *Recurrent Neural Network* and *Convolutional Neural Network*. Each of them will face a classification problem: the correct recognition of handwritten numbers from 0 to 9. Therefore, after having introduced the MNIST dataset, the three algorithms will be implemented exploiting the *Object Oriented Programming* (OOP): every model will be formalized as a python `class`. Finally it will be possible to identify which of them achieves the best accuracy on MNIST dataset.

3.1 MNIST dataset

The MNIST (Mixed National Institute of Standards and Technology) database is a large dataset of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It was created by re-mixing the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments. Furthermore, the black and white images from NIST were normalized to fit into a 20×20 pixel bounding box and anti-aliased, which introduced grayscale levels.
The MNIST data is split into three parts: 55,000 data points of training data (`mnist.train`), 10,000 points of test data (`mnist.test`), and 5,000 points of validation data (`mnist.validation`). This last one is used to compare different approaches we have trained before and so to choose the one with the best performance. This split is very important: it is essential in machine learning that we have separate data

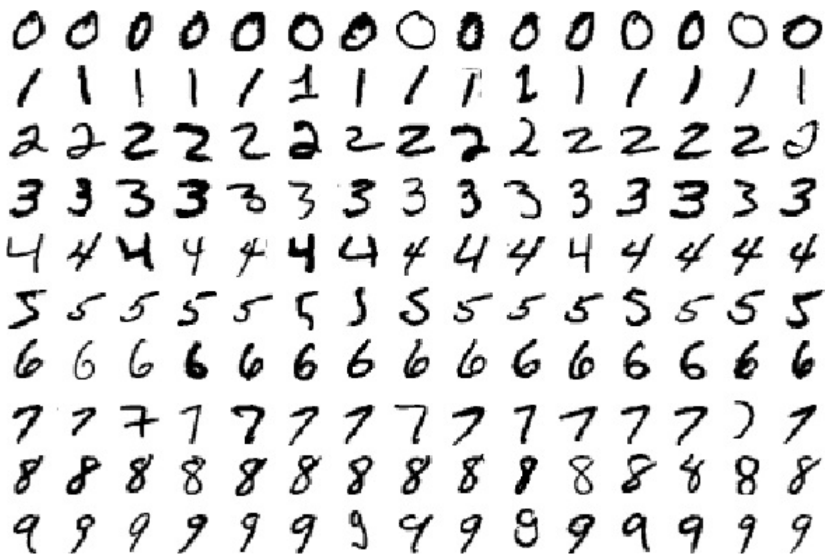


Figure 3.1: Part of the handwritten digits included in the MNIST dataset. (From https://kuanhoong.wordpress.com)

which we do not learn from so that we can make sure that what we have learned actually generalizes. Half of the training, test and validation sets were taken from NIST's training dataset, while the remaining half parts were taken from NIST's testing dataset. There have been a number of scientific papers on attempts to achieve the lowest error rate. The original creators of the database keep a list of some of the methods tested on it. In their original paper, they use a *Support Vector Machine* (SVM) to get an error rate of 0.8 percent.

The set of images in the MNIST database is a combination of two of NIST's databases: *Special Database 1* and *Special Database 3*. Both oh them consist of digits written by high school students and employees of the *United States Census Bureau*, respectively.

Every MNIST data point has two parts: an image of a handwritten digit and a corresponding label. The images will call `X` and the labels `Y`. Both the training set and test set contain images and their corresponding labels. For instance the training images are `mnist.train.images` and the training labels are `mnist.train.labels`. Every image is composed by 28×28 pixels. It is possible to interpret them as a big array of numbers. Thus the MNIST database is nothing but a set of 784-dimensional arrays, whose entries represent pixels magnitude, bounded between 0 and 1. Besides, it is convenient to formalize the label as 10-dimensional arrays whose entries are all 0s apart from one which assumes value 1. For instance, the number 2 would be $[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$. Moreover the labels are called *one-hot vectors* and it is possible to perform this representation through the command `mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)`.

Thus the result is that `mnist.train.images` and `mnist.train.labels` are two tensors with shape $[55000, 784]$ and $[55000, 10]$ respectively.

3.2 Softmax regression

As illustrated in the previous section, every image in MNIST is a handwritten digit between zero and nine. So there are only ten possible things that a given image can be. The goal is to be able to look at an image and give the probabilities for it being each digit. This is a classic case where a *softmax regression* is a natural, simple model.
Softmax regression (or multinomial logistic regression) is a generalization of *logistic regression* to the case where we want to handle multiple classes. In logistic regression we assume that the labels are binary: $y_i \in \{0, 1\}$. Softmax regression allows us to handle $y_i \in \{1, \ldots, K\}$ where K is the number of classes.
In logistic regression, we had a training set $\{(x_1, y_1), \ldots, (x_m, y_m)\}$ of m labelled examples, where the input features are $x_i \in \Re^n$. Moreover, we are in the binary classification setting, so the labels are $y_i \in \{0, 1\}$. Consequently the hypothesis is:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}} \tag{3.1}$$

The model parameters θs are trained to minimize the cost function:

$$J(\theta) = -\sum_{i=1}^{m} y_i \log(h_\theta(x_i)) + (1 - y_i) \log(1 - h_\theta(x_i)) \tag{3.2}$$

In the softmax regression setting, we are interested in multi-class classification (as opposed to only binary classification), and so the label y can take on K different values, rather than only two. Thus, in the training set $\{(x_1, y_1), \ldots, (x_m, y_m)\}$, we have that $y_i \in \{1, 2, \ldots, K\}$. For instance, in the MNIST digit recognition task, we would have $K = 10$ different classes.
Given a test input x, we want our hypothesis to estimate the probability that $P(y = k|x)$ for each value of $k = \{1, \ldots, K\}$. More precisely we want to estimate the probability of the class label taking on each of the K different possible values. Thus, our hypothesis will output a K-dimensional vector (whose elements sum to 1) giving us K estimated probabilities. Concretely, the hypothesis $h_\theta(x)$ becomes:

$$h_{\theta_i}(x) = P(y = i|x; \theta) = \frac{1}{\sum_{j=1}^{K} e^{(\theta_j)^T x}} e^{(\theta_i)^T x} \tag{3.3}$$

with $i = (1, ..., K)$ and $\theta_i \in \Re^n$ are the parameters of the model. The term:

$$\sum_{j=1}^{K} e^{(\theta_j)^T x} \tag{3.4}$$

normalizes the distribution, therefore it sums to one.
For convenience, θ denotes all the parameters of our model. When implementing softmax regression, it is usually convenient to represent θ as a n-by-K matrix obtained by concatenating $\theta_1, \theta_2, \ldots, \theta_K$ into columns.

A commonly used cost function for softmax regression is the *cross entropy*, described as follows. In the equation below, $1\{\cdot\}$ is the *indicator function*, so that $1\{a\ true\ statement\} = 1$ and $1\{a\ false\ statement\} = 0$. For example, $1\{2+2=4\}$ evaluates to 1; whereas $1\{1+1=5\}$ evaluates to 0. Thus, the cost function can be written as follows:

$$J(\theta) = -\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y_i = k\} \log(P(y = i|x;\theta)). \tag{3.5}$$

In TensorFlow the cross entropy loss function is represented by the instruction: `tf.nn.softmax_cross_entropy_with_logits`.
It is impossible to solve for the minimum of $J(\theta)$ analytically, and thus as usual we resort to an iterative optimization algorithm. In the following examples, we are going to use the *Adam*, a method for the stochastic optimization described in the next chapter.

3.3 Multilayer Perceptron

A multilayer perceptron (MLP) is a feedforward artificial neural network model that maps sets of input data onto a set of appropriate outputs. Feedforward means that data flows in one direction from input to output (forward). It consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one: this is the reason why we are going to analyse the application of MLP to the MNIST dataset in this chapter, indeed it is a way to approach the concept of *deep network* which we will exploit in the first part of the machine proposed in the thesis.
MLP is a modification of the standard linear perceptron and can distinguish data that are not linearly separable. The term "multilayer perceptron" often causes confusion. It is argued the model is not a single perceptron that has multiple layers. Rather, it contains many perceptrons that are organised into layers, leading some to believe that a more fitting term might therefore be "multilayer perceptron network". Moreover, these "perceptrons" are not really perceptrons in the strictest possible sense, as true perceptrons are a special case of artificial neurons that use a threshold activation function such as the Heaviside step function, whereas the artificial neurons in a multilayer perceptron are free to take on any arbitrary activation function. Consequently, whereas a true perceptron performs binary classification, a neuron in a multilayer perceptron is free to either perform classification or regression, depending upon its activation function. The one utilized in this example is the *Rectified Linear Unit* (ReLU) activation function. It is defined as:

$$f(x) = max(0, x) \tag{3.6}$$

where x is the input to a neuron.
The structure of the implemented MLP is shown in Fig. 3.2 . It is composed by three hidden layers, each with 500 units and one output layer. Every layer is fully connected to the next one. Because the model is trained in minibatch mode, the shapes of the tensors `X` and `Y` are $[batch\ size, 784]$ and $[batch\ size, 10]$ respectively.

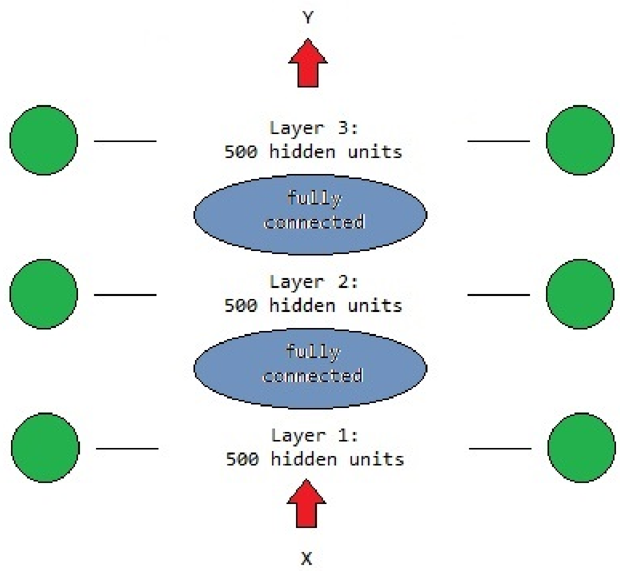


Figure 3.2: In this figure the MLP structure is showed. The input tensor `X` has shape "batch size"-by-784 while the output tensor Y has shape "batch size"-by-10.

As aforesaid, each model is formalized as a python `class`.In the MLP case, this class takes the form displayed in Fig. 3.3 . It is well worth to notice how the model is evaluated. Firstly it is figured out where the correct label has been predicted. The function `tf.argmax()` is extremely useful. It returns the index of the highest entry in a tensor along some axis. For instance, `tf.argmax(self.output, 1)` is the label the model thinks is most likely for each input, while `tf.argmax(Y, 1)` is the correct label. We can use `tf.equal` to check if our prediction matches the truth. This gives us a list of booleans. To determine what fraction are correct, we cast to floating point numbers and then take the mean as done in `self.accuracy`. For example, `[True, False, True, True, True]` would become `[1,0,1,1,1]` and so the resulting accuracy would be 0.80.
Using the OOP it is possible to formalize the training and testing phases as two *methods* of the class (see Fig. 3.4) . It is a very useful practice. Indeed after having created an instance of the class, it is possible to train and test separately the model simply invoking the relative methods (see Fig. 3.14).
Finally, the method `get_parameters` is defined to observe the values assumed by the weights and the biases of each layer.

```python
'''
Application of a multi-layer perceptron to the MNIST dataset.
The machine is composed by three hidden layers, each with 500 units,
and one output layer. The pixels of every image (width=28, height=28 ==> 784 pixels)
are flattened into a row. Therefore they are stored in the tensor X
(that has shape 'batch size' by '784').
Hyperparameters:
 - input_dim: 784 pixels;
 - epochs: how long the MLP is trained;
 - batch_size: 128. This number is chosen to cite those familiar numbers 32, 64, 128, 256... ;
 - learning_rate: it allows to change the default learning rate of the optimizer;
 - n_classes: numbers from 0 to 9.
'''

import tensorflow as tf

class MLP:

    def __init__(self, input_dim, n_nodes_hl, epochs = 10, batch_size = 128, learning_rate = 0.01, n_classes = 10):

        # Hyperparameters

        self.input_dim = input_dim
        self.epochs = epochs
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.n_classes = n_classes

        # Input and target placeholders

        X = tf.placeholder("float", [None, self.input_dim])
        Y = tf.placeholder("float")

        '''
        Weights and biases of the layers are formalized as dictionaries.
        The weights of a layer represent a tensor of shape 'number of units of
        the previous layer' by 'number of units of the layer'.
        The biases of a layer represent a tensor of shape 'number of units of the layer'.
        The command tf.random_normal fill the tensors with random normal values.
        The rectified linear activation function is applied to the output of
        every hidden layer. It computes max(features, 0).
        '''

        with tf.name_scope('MLP'):
            weights = {'weights_hl1': tf.Variable(tf.random_normal([self.input_dim, n_nodes_hl[0]])),
                       'weights_hl2': tf.Variable(tf.random_normal([n_nodes_hl[0], n_nodes_hl[1]])),
                       'weights_hl3': tf.Variable(tf.random_normal([n_nodes_hl[1], n_nodes_hl[2]])),
                       'weights_out_l': tf.Variable(tf.random_normal([n_nodes_hl[2], self.n_classes]))
                       }

            biases = {'biases_hl1': tf.Variable(tf.random_normal([n_nodes_hl[0]])),
                      'biases_hl2': tf.Variable(tf.random_normal([n_nodes_hl[1]])),
                      'biases_hl3': tf.Variable(tf.random_normal([n_nodes_hl[2]])),
                      'biases_out_l': tf.Variable(tf.random_normal([self.n_classes]))
                      }

            output_layer_1 = tf.add(tf.matmul(X, weights['weights_hl1']), biases['biases_hl1'])
            output_layer_1 = tf.nn.relu(output_layer_1)

            output_layer_2 = tf.add(tf.matmul(output_layer_1, weights['weights_hl2']), biases['biases_hl2'])
            output_layer_2 = tf.nn.relu(output_layer_2)

            output_layer_3 = tf.add(tf.matmul(output_layer_2, weights['weights_hl3']), biases['biases_hl3'])
            output_layer_3 = tf.nn.relu(output_layer_3)

            output = tf.matmul(output_layer_3, weights['weights_out_l']) + biases['biases_out_l']

        self.X = X
        self.Y = Y
        self.weights = weights
        self.biases = biases
        self.output = output

        # Loss function and optimizer

        self.cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.output, Y))
        self.optimizer = tf.train.AdamOptimizer(self.learning_rate).minimize(self.cost)

        self.correct = tf.equal(tf.argmax(self.output, 1), tf.argmax(self.Y, 1))
        self.accuracy = tf.reduce_mean(tf.cast(self.correct, 'float'))

        # Needed to save the session

        self.saver = tf.train.Saver()
```

Figure 3.3: MLP `class`: `__init__`.

```
def train(self, data):

    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())

        for epoch in range(self.epochs):
            epoch_loss = 0
            for _ in range(int(data.train.num_examples/self.batch_size)):
                x, y = data.train.next_batch(self.batch_size)
                _, c = sess.run([self.optimizer, self.cost], feed_dict ={ self.X: x, self.Y: y})
                epoch_loss += c
            print('Epoch', (epoch+1), 'completed out of', self.epochs, 'loss:', epoch_loss)

        self.saver.save(sess, './MLP.ckpt')
        print('Accuracy', self.accuracy.eval({self.X: data.train.images, self.Y: data.train.labels}))

def test(self, data):

    with tf.Session() as sess:
        self.saver.restore(sess, './MLP.ckpt')
        print('Accuracy', self.accuracy.eval({self.X: data.test.images, self.Y: data.test.labels}))
```

Figure 3.4: Train and test methods. As the code shows, it is made use of the `tf.train.Saver` object. In this way, it is possible to test the algorithm without retraining it but rather restoring the model from the checkpoint file.

```
def get_parameters(self):

    with tf.Session() as sess:
        self.saver.restore(sess, './MLP.ckpt')
        weights, biases = sess.run([self.weights, self.biases])
        print('HIDDEN LAYER 1: \n',
              'weights: \n', weights['weights_hl1'], '\n',
              'biases: \n', biases['biases_hl1'], '\n',
              'HIDDEN LAYER 2: \n',
              'weights: \n', weights['weights_hl2'], '\n',
              'biases: \n', biases['biases_hl2'], '\n',
              'HIDDEN LAYER 3: \n',
              'weights: \n', weights['weights_hl3'], '\n',
              'biases: \n', biases['biases_hl3'], '\n'
              'OUTPUT LAYER: \n',
              'weights: \n', weights['weights_out_l'], '\n',
              'biases: \n', biases['biases_out_l'], '\n'
              )

    return weights, biases
```

Figure 3.5: This method eases the understanding of what is happening in the algorithm.

3.4 Convolutional neural network

Convolutional Networks (LeCun, 1989), also known as *convolutional neural networks* or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels.
Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called *convolution*. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks which use convolution in place of general matrix multiplication in at least one of their layers. In addition to it, most of the CNNs employs the *pooling*. Starting from convolution, these two operations will be described below.

3.4.1 Convolution

Usually, the used operation in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields such as engineering or pure mathematics.
In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, consider the following examples.
Suppose we are tracking the location of a spaceship with a laser sensor. The laser sensor provides a single output $x(t)$, the position of the spaceship at time t. Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.
Now suppose that the laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da \tag{3.7}$$

This operation is called **convolution**. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t) \tag{3.8}$$

In this example, w needs to be a valid probability density function, or the output is not a weighted average. Also, w needs to be 0 for all negative arguments,or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to the considered example though. In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.

In CNN terminology, the first argument (in this example, the function x) to the convolution is often referred to as the *input* and the second argument (in this example, the function w) as the *kernel*. The output is sometimes referred to as the *feature map*.
In this example, the idea of a laser sensor that can provide measurements at every instant in time is not realistic. Usually, when we work with data on a computer, time will be made discrete, and the sensor will provide data at regular intervals. In this case, it might be more realistic to assume that the laser provides a measurement once per second. The time index t can then take on only integer values. If we now assume that x and w are defined only on integer t, we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{3.9}$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, it is usually assumed that these functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.
Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a 2-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3.10}$$

In Fig. 3.6 it is shown an example of convolution applied to a 2-dimensional tensor.

3.4.2 Pooling

A typical layer of a convolutional network consists of three stages (see Fig. 3.7).
In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a non linear activation function, such as the rectified linear activation function. This stage is sometimes called the *detector stage*. In the third stage, we use a *pooling function* to modify the output of the layer further.
A pooling function replaces the output of the network at a certain location with a summary statistic of the nearby outputs. For example, the *max pooling* operation reports the maximum output within a rectangular neighbourhood. Other popular pooling functions include the average of a rectangular neighbourhood, the L2 norm of a rectangular neighbourhood, or a weighted average based on the distance from the central pixel. In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation

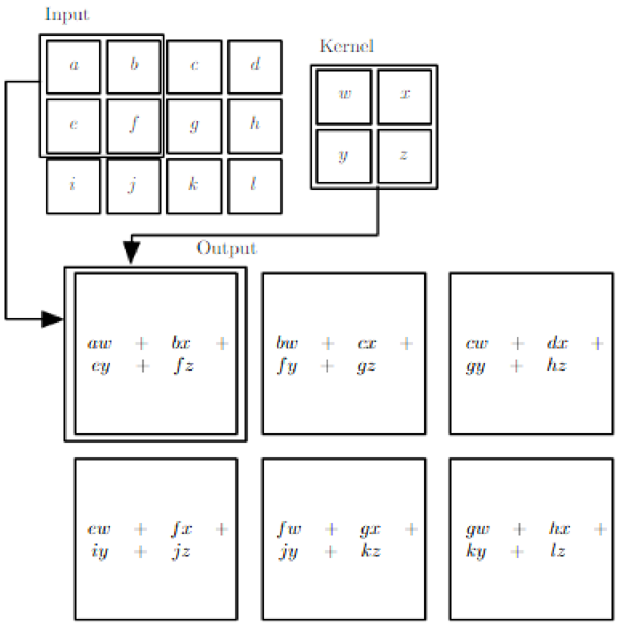


Figure 3.6: An instance of 2-D convolution. The output has been restricted to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts.

means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is. For example, when determining whether an image contains a bicycle, we do not need to know the location of the wheels with pixel-perfect accuracy, we just need to know that there is a wheel on both the extremity of the bicycle.

3.4.3 The model

To implement the CNN we are going to start from the MLP code and then we will introduce some modifications. Much of our code structure is different, but we tried to keep the variable and parameter names that matter the same as the MLP code. The algorithm implemented consists of two convolutional layers, a densely connected layer and in the end an output layer.
Consider Fig.3.8 . First of all it is well worth to notice that convolution and pooling operations are already defined in TensorFlow. To invoke them, we have just to write the commands:

- `tf.nn.conv2d()`;
- `tf.nn.max_pool()`.

Moreover, TF gives us a lot of flexibility in convolution and pooling operations. Indeed we can decide how to handle the boundaries or what should be the stride size. The `strides` parameter dictates the movement of the window. So in this case, we just move 1 pixel at a time for the convolutions, and 2 at a time for the max

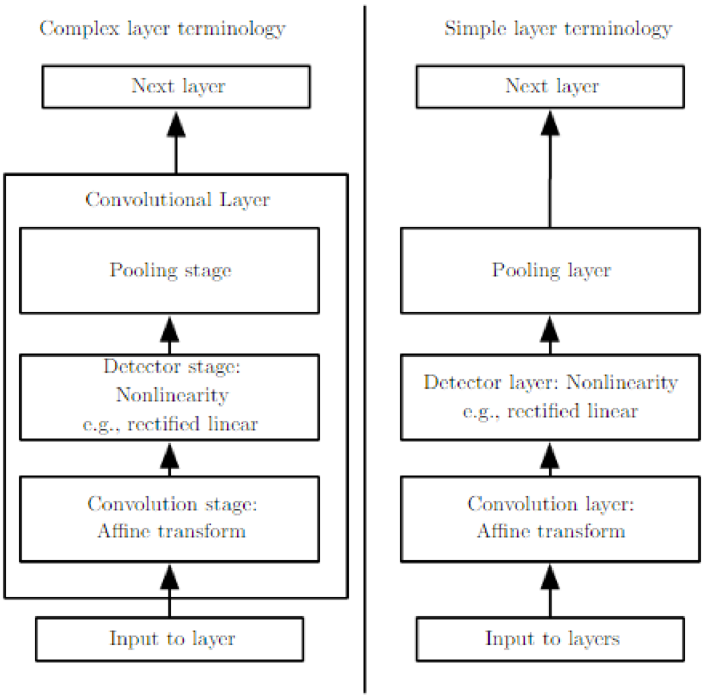


Figure 3.7: Stages of a typical CNN layer. There are two commonly used sets of terminology for describing these layers. *Left)* In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many "stages". In this terminology, there is a one-to-one mapping between kernel tensors and network layers. *Right)* In this terminology, the convolutional network is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every "layer" has parameters. (From 34)

pooling operation. The `ksize` parameter is the size of the pooling window. In our case, we are choosing a 2×2 pooling window for pooling.
We can now implement the first layer. It consists of a convolution, followed by max pooling operation. The convolution computes "number of nodes in the first layer" features for each 5×5 patch. Its weight tensor has a shape of $[5, 5, 1, n\_nodes\_hl[0]]$. The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels. We also have a bias vector with a component for each output channel.
To apply the layer, we first reshape `X` to a 4-dimensional tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels. We then convolve the input image with the weight tensor, add the bias, apply the ReLU function, and finally the max pooling. IN this way, `max_pool` 2×2 method reduces the image size to 14×14.
Thus we move to the second convolutional layer. It takes as input the output of the previous layer and produces `n_nodes_hl[1]` features. Moreover the max pooling operation reduces the image size to 7×7.
Now that the image size has been reduced to 7×7, we add a fully-connected layer with "number of nodes in the third layer" neurons to allow processing on the entire image. Firstly we need to reshape the tensor from the pooling layer into a batch of vectors, then we compute the output and apply a ReLU. Finally, we add an output layer.
At this point we can train and test our model (see Fig. 3.9). Thus the usual `train`, `test` and `get_parameters` methods are defined. It is well worth to notice that also during training and evaluating accuracy we need to reshape the input images.

```python
'''
Application of a Convolutional Neural Network to the MNIST dataset.
The machine is composed by two convolutional layers, one fully connected
layer and an output layer. The pixels of every image (width=28, height=28 ==> 784 pixels)
are flattened into a row. Therefore they are stored in the tensor X
(that has shape 'batch size' by '784').
Hyperparameters:
 - input_dim: 784 pixels;
 - epochs: how long the CNN is trained;
 - batch_size: 128. This number is chosen to cite those familiar numbers 32, 64, 128, 256... ;
 - learning_rate: it allows to change the default learning rate of the optimizer;
 - n_classes: numbers from 0 to 9.
'''

import tensorflow as tf
import numpy as np

class Convolutional_Net:

    def __init__(self, input_dim, n_nodes_hl, epochs = 10, batch_size = 128, learning_rate = 0.01, n_classes = 10):

        # Hyperparameters

        self.input_dim = input_dim
        self.epochs = epochs
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.n_classes = n_classes

        # Input and target placeholders

        X = tf.placeholder("float", [None, input_dim]) #the pixels are on a row
        Y = tf.placeholder("float")

        with tf.name_scope('Convolutional_Neural_Network'):

            '''
            Weights and biases of the layers are formalized as dictionaries.
            First convolutional layer: 5 by 5 convolution, it takes 1 input
                                       and returns 'n_nodes_hl[0]' outputs.
            Second convolutional layer: 5 by 5 convolution, 'n_nodes_hl[0]' inputs
                                        and 'n_nodes_hl[1]' outputs.
            Densely connected layer: it is added once that the image size is reduced to 7x7.
                                     Therefore the inputs are '7*7*n_nodes_hl[1]' and the outputs
                                     'n_nodes_hl[2]'.
            Output layer: 'n_nodes_hl[2]' inputs, 'n_classes' targets.
            '''

            weights = {'weights_convl1': tf.Variable(tf.random_normal([5, 5, 1, n_nodes_hl[0]])),
                       'weights_convl2': tf.Variable(tf.random_normal([5, 5, n_nodes_hl[0], n_nodes_hl[1]])),
                       'weights_fully_connected': tf.Variable(tf.random_normal([7*7*n_nodes_hl[1], n_nodes_hl[2]])),
                       'weights_output': tf.Variable(tf.random_normal([n_nodes_hl[2], self.n_classes]))}

            biases = {'b_convl1': tf.Variable(tf.random_normal([n_nodes_hl[0]])),
                      'b_convl2': tf.Variable(tf.random_normal([n_nodes_hl[1]])),
                      'b_fully_connected': tf.Variable(tf.random_normal([n_nodes_hl[2]])),
                      'b_output': tf.Variable(tf.random_normal([self.n_classes]))}

            '''
            Reshape X to 4d tensor to apply the first convolutional layer.
            The second and the third dimensions correspond to the image width
            and height, the final dimension to the number of color channels.
            '''
            X = tf.reshape(X, shape = [-1, 28, 28, 1])

            '''
            Then, the following operations are applied:
             - tf.nn.conv2d: it computes a 2d convolution, given a 4d tensor, through a sliding
                             window (in this case a stride of 1 pixel for each dimension);
             - compute the output and apply the rectified linear activation function;
             - tf.nn.max_pool: it performs the max pooling over 2x2 blocks. This operation
                               halves the image size (28x28 =>(conv1)=> 14x14);
             - compute the output of the densely connected layer and apply the rectified
               linear activation function. This layer allows processing on the entire image;
             - compute the final output.
            '''

            conv1 = tf.nn.relu(tf.nn.conv2d(X, weights['weights_convl1'], strides=[1, 1, 1, 1], padding='SAME') + biases['b_convl1'])
            conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

            conv2 = tf.nn.relu(tf.nn.conv2d(conv1, weights['weights_convl2'], strides=[1, 1, 1, 1], padding='SAME') + biases['b_convl2'])
            conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

            fully_connected = tf.reshape(conv2, [-1, 7*7*n_nodes_hl[1]])
            fully_connected = tf.nn.relu(tf.matmul(fully_connected, weights['weights_fully_connected']) +
                                         biases['b_fully_connected'])

            output = tf.matmul(fully_connected, weights['weights_output']) + biases['b_output']

        self.X = X
        self.Y = Y
        self.weights = weights
        self.biases = biases
        self.output = output

        # Loss function and optimizer

        self.cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.output, self.Y))
        self.optimizer = tf.train.AdamOptimizer(self.learning_rate).minimize(self.cost)

        self.correct = tf.equal(tf.argmax(self.output, 1), tf.argmax(self.Y, 1))
        self.accuracy = tf.reduce_mean(tf.cast(self.correct, 'float'))

        # Needed to save the session

        self.saver = tf.train.Saver()
```

```
def train(self, data):

    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())

        for epoch in range(self.epochs):
            epoch_loss = 0
            for _ in range(int(data.train.num_examples/self.batch_size)):
                x, y = data.train.next_batch(self.batch_size)

                # Set the correct shape of the inputs
                x = x.reshape((self.batch_size, 28, 28, 1))

                _, c = sess.run([self.optimizer, self.cost], feed_dict ={ self.X: x, self.Y: y})
                epoch_loss += c

            print('Epoch', (epoch+1), 'completed out of', self.epochs, 'loss:', epoch_loss)

        self.saver.save(sess, './ConvNet.ckpt')

'''
Test the algorithm on a smaller test set (1500 images) since a convolutional neural
network requires a lot of computational resources.
'''

def test(self, data):
    x, y = data.test.next_batch(1500)

    with tf.Session() as sess:
        self.saver.restore(sess, './ConvNet.ckpt')
        print('Accuracy', self.accuracy.eval({self.X: x.reshape((-1, 28, 28, 1)), self.Y: y}))

def get_parameters(self):

    with tf.Session() as sess:
        self.saver.restore(sess, './ConvNet.ckpt')
        weights, biases = sess.run([self.weights, self.biases])
        print('CONVOLUTIONAL LAYER 1: \n',
              'weights: \n', weights['weights_convl1'], '\n',
              'biases: \n', biases['b_convl1'], '\n',
              'CONVOLUTIONAL LAYER 2: \n',
              'weights: \n', weights['weights_convl2'], '\n',
              'biases: \n', biases['b_convl2'], '\n',
              'FULLY CONNECTED LAYER: \n',
              'weights: \n', weights['weights_fully_connected'], '\n',
              'biases: \n', biases['b_fully_connected'], '\n'
              'OUTPUT LAYER: \n',
              'weights: \n', weights['weights_output'], '\n',
              'biases: \n', biases['b_output'], '\n'
              )

    return weights, biases
```

Figure 3.9: `train`, `test` and `get_parameters` methods.

3.5 Recurrent neural network

Recurrent neural networks (RNNs) (Rumelhart et al., 1986) are a family of neural networks for processing sequential data.
While a convolutional network is a neural network that is specialized for processing a grid of values `X` such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\{x_1, ..., x_n\}$. Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length. This is the reason why we analyse the application of this model to the MNIST dataset in this chapter: it is a simple way to introduce a recurrent neural network which will be a fundamental part of the machine proposed in this thesis.
In order to go from multilayer networks to recurrent networks, we need to take advantage of one of the early ideas found in machine learning and statistical models of the 1980s: sharing parameters across different parts of a model.
Parameter sharing makes possible the extension and the application of the model to examples of different forms and then the generalization across them. If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time. Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence.
For example, consider the two sentences "I went to Rome in 2009" and "In 2009, I went to Rome". If we ask a machine learning model to read each sentence and extract the year in which the narrator went to Rome, we would like it to recognize the year 2009 as the relevant piece of information, whether it appears in the sixth word or the second word of the sentence. Suppose that we trained a feedforward network that processes sentences of fixed length. A traditional fully connected feedforward network would have separate parameters for each input feature, so it would need to learn all of the rules of the language separately at each position in the sentence. By comparison, a recurrent neural network shares the same weights across several time steps. Recurrent neural networks share parameters in a different way. Each member of the output is a function of the previous members of the output and it is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph.
Consider RNNs as operating on a sequence containing vectors x_t with $i \in \{1, ..., n\}$. In practical application, recurrent networks usually operate on minibatches of such sequences, with a different sequence length n for each member of the minibatch. It has been omitted the minibatch indices to simplify notation. Moreover, the necessary time step index does not literally refer to the passage of time in the real world. Sometimes it refers only to the position in the sequence. Besides a RNN may also be applied in two dimensions across spatial data, indeed in this section it will be

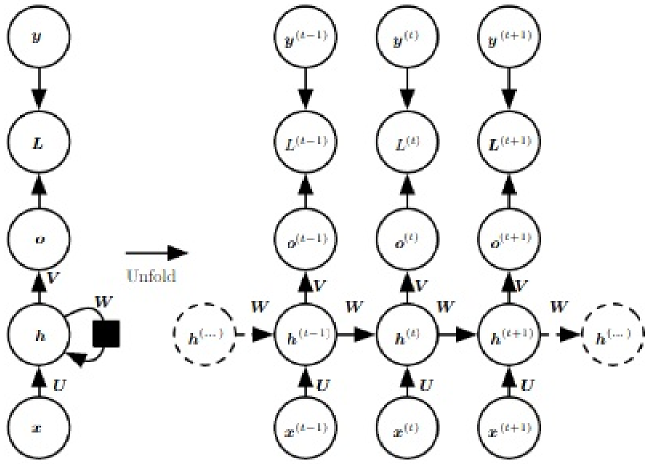


Figure 3.10: Rolled and unfolded version of RNN computational graph. It is possible to understand how to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output o values.
A loss L measures how far each o is from the corresponding training target y. The RNN has input to hidden connections parametrized by a weight matrix U, hidden-to-hidden recurrent connections parametrized by a weight matrix W, and hidden-to-output connections parametrized by a weight matrix V. (From 36)

applied to MNIST dataset.
By analysing the computational graph of a RNN it is possible to notice interesting characteristics of this kind of networks. A typical computational graph is illustrated in Fig. 3.10 .
Two possible representation of RNN are showed:

- on the left, we can observe the rolled diagram and its loss drawn with recurrent connections;
- on the right, the same diagram seen as a time-unfolded computational graph, where each node is now associated with one particular time instance.

If recurrence makes the rolled RNN computational graph seems kind of mysterious, the unfolded view shows that RNN is not tremendously different from a normal neural network. Indeed, it can be thought as multiple copies of the same network, each passing a message to a successor.
It is possible to represent the unrolled recurrence after t steps with a function g_t:

$$h_t = g_t(x_t, x_{t-1}, ..., x_2, x_1) = f(h_{t-1}, x_t; \theta) \tag{3.11}$$

The function g_t takes the whole past sequence $\{x_t, x_{t-1}, ..., x_2, x_1\}$ as input and produces the current state, but the unfolded recurrent structure allows us to factorize g_t into repeated application of a function f. The unfolding process thus introduces two major advantages:

- regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states;

- it is possible to use the same transition function f with the same parameters at every time step.

These two factors make possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model g_t for all possible time steps. Learning a single, shared model allows generalization to sequence lengths that did not appear in the training set, and allows the model to be estimated with far fewer training examples than would be required without parameter sharing. Both the recurrent graph and the unrolled graph have their uses. The recurrent graph is succinct. The unfolded graph provides an explicit description of which computations to perform. Moreover, the unfolded graph helps to illustrate the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients) by explicitly showing the path along which this information flows.
Even though RNNs represent a powerful class of model, they have a problem related to the learning of long-term dependencies. The basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode(rarely, but with much damage to the optimization). By introducing the *long short-term memory* (LSTM) it is possible to overcome it. A complete description of this problem and a more detailed illustration of LSTM cell will be provided in the next chapter. For the moment believe in it and let us focus on how to apply the RNN with LSTM cell to the MNIST database.

3.5.1 The model

To implement the RNN with LSTM cell we are going to start again from the MLP code and then we will introduce some modifications.
First of all it is necessary to import the RNN model/cell from TF through the command:
`from tensorflow.python.ops import rnn, rnn_cell`.
In the following implementation we will use the `class`:
`tf.nn.rnn_cell.BasicLSTMCell`.
It is based on Zaremba et al., 2015 (18). The main contribute of this paper was a recipe for applying a widely known technique, the *dropout*, to LSTMs in a way that successfully reduces overfitting.
Dropout was introduced by Srivastava in 2013 (15) and it rapidly becomes the most powerful regularization method for large feedforward neural networks. Indeed, even though large deep neural network are very powerful machine learning systems, overfitting is a serious problem in such networks. Moreover, large networks are slow to use as well, making it difficult to deal with overfitting by combining the predictions of many large neural networks at test time. The key idea of dropout, is to randomly drop units (along with their connections) from the neural network during training, sampling from an exponential number of different "thinned" network. At the test phase, it is simply used a single "unthinned" network that has smaller weights.
Unfortunately this first version of dropout does not work well with RNNs. By modifying its application, Zaremba et al. were able to successfully apply dropout to

RNNs as already said. The main idea was to apply the dropout operator only to the non-recurrent connections. Indeed, it is fundamental that the units remember events that occured many time steps in the past. In this way, the LSTM "memory" remains untouched.
In addition we need to define some additional quantities. Indeed to classify images using a recurrent neural network, we consider every image row as a sequence of pixels. Since MNIST image shape is 28-by-28 pixels, we will define 28 sequences (`seq_size`) each consisting of 28 pixels (`input_dim`). Furthermore we get some modifications to the input tensor `X` to satisfy the structure that TensorFlow `rnn_cell` model requires (see Fig. 3.13).
Thus the usual `train`, `test` and `get_parameters` methods are defined. As in the CNN examples, it is well worth to notice that also during training and evaluating accuracy we need to reshape the input images. As we will see when building the machine in the next chapter, this code will represent a starting point for the construction of the second part of the architecture proposed in the thesis. Indeed, we will only need to make some modifications to adapt the code to our future task.
To conclude this chapter, in the next section we are going to compare the results of the three examples.

```python
'''
Application of a Recurrent Neural Network (with LSTM cell) to the MNIST dataset.
Every image is divided in 28 sequences, each one of 28 pixels. Therefore they are
stored in a tensor X of shape 'batch size' by 'sequence size' by 'input dimension'.
Hyperparameters:
 - input_dim: 784 pixels;
 - seq_size: number of sequences;
 - hidden_dim: RNN size;
 - epochs: how long the RNN is trained;
 - batch_size: 128. This number is chosen to cite those familiar numbers 32, 64, 128, 256... ;
 - learning_rate: it allows to change the default learning rate of the optimizer;
 - n_classes: numbers from 0 to 9;
 - keep_prob: it makes possible an optional drop out one of the most common
   regularization tecnique. In this case it is set to 1.0 (consequently the
   drop out is not applied).
'''

import tensorflow as tf
from tensorflow.python.ops import rnn, rnn_cell

class RNN_LSTM:

    def __init__(self, input_dim, seq_size, n_classes = 10, hidden_dim = 128, learning_rate = 0.01, batch_size = 128,
                 epochs = 5, keep_prob = 1.0):
        # Hyperparameters

        self.input_dim = input_dim
        self.seq_size = seq_size
        self.n_classes = n_classes
        self.hidden_dim = hidden_dim
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.epochs = epochs
        self.keep_prob = keep_prob

        # Input and target placeholders ###

        X = tf.placeholder("float", [None, self.seq_size, self.input_dim])
        Y = tf.placeholder("float")

        # Weights & biases ###

        weights = {'weights' : tf.Variable(tf.random_normal([self.hidden_dim, self.n_classes]))}
        biases = {'biases' : tf.Variable(tf.random_normal([self.n_classes]))}

        self.X = X
        self.Y = Y
        self.weights = weights
        self.biases = biases
        self.output = self.lstm()

        # Loss function and optimizer

        self.cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.output, self.Y))
        self.optimizer = tf.train.AdamOptimizer().minimize(self.cost)

        self.correct = tf.equal(tf.argmax(self.output, 1), tf.argmax(self.Y, 1))
        self.accuracy = tf.reduce_mean(tf.cast(self.correct, 'float'))

        # Needed to save the session

        self.saver = tf.train.Saver()
```

Figure 3.11: RNN `class`: `__init__`.

```python
def lstm(self):
    '''
    Prepare the input shape for the lstm. Actual shape is (batch_size, seq_size, input_dim),
    the required shape is 'seq_size' tensors list of shape (batch_size, n_input)

    It is fundamental to copy self.X in a new tensor X since otherwise
    the following transformations modify the shape of self.X, raising the error:
    unhashable type: 'list'.
    '''

    X = self.X
    # Permuting batch_size and seq_size
    X = tf.transpose(X, [1, 0, 2])

    # Reshaping to (seq_size*batch_size, input_dim)
    X = tf.reshape(X, [-1, self.input_dim])

    # Split to get a list of 'seq_size' tensors of shape (batch_size, input_dim)
    X = tf.split(0, self.seq_size, X)

    # Create lstm and add the dropout

    lstm_cell = rnn_cell.BasicLSTMCell(self.hidden_dim)
    lstm_cell = rnn_cell.DropoutWrapper(lstm_cell, input_keep_prob=self.keep_prob, output_keep_prob=self.keep_prob)
    outputs, states = rnn.rnn(lstm_cell, X, dtype=tf.float32)

    output = tf.matmul(outputs[-1], self.weights['weights']) + self.biases['biases']

    return output
```

Figure 3.12: Method defininig the LSTM.

```python
def train(self, data):

    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())

        for epoch in range(self.epochs):
            epoch_loss = 0
            for _ in range(int(data.train.num_examples/self.batch_size)):
                batch_x, batch_y = data.train.next_batch(self.batch_size)

                # Set the correct shape of the inputs
                batch_x = batch_x.reshape((self.batch_size, self.seq_size, self.input_dim))

                _, c = sess.run([self.optimizer, self.cost], feed_dict ={ self.X: batch_x, self.Y: batch_y})
                epoch_loss += c

            print('Epoch', (epoch+1), 'completed out of', self.epochs, 'loss:', epoch_loss)

        save_path = self.saver.save(sess, './lstm.ckpt')
        print('Accuracy', self.accuracy.eval({self.X: data.train.images.reshape((-1, self.seq_size, self.input_dim)),
                                              self.Y: data.train.labels}))
def test(self, data):

    with tf.Session() as sess:
        self.saver.restore(sess, './lstm.ckpt')

        print('Accuracy', self.accuracy.eval({self.X: data.test.images.reshape((-1, self.seq_size, self.input_dim)),
                                              self.Y: data.test.labels}))
def get_parameters(self):
    with tf.Session() as sess:
        self.saver.restore(sess, './lstm.ckpt')
        weights, biases = sess.run([self.weights, self.biases])
        print('LSTM: \n',
              'weights: \n', weights['weights'], '\n',
              'biases: \n', biases['biases'], '\n'
              )
    return weights, biases
```

Figure 3.13: `train`, `test` and `get_parameters` methods.

```python
from tensorflow.examples.tutorials.mnist import input_data
from RNN_OOP import RNN_LSTM
from MLP_OOP import MLP
from Conv_Net_OOP import Convolutional_Net

### READ THE INPUTS AND SET THE LABELS AS ONE HOT VECTORS ###
data = input_data.read_data_sets("/tmp/data/", one_hot=True)


### Multi-Layer Perceptron ###

MLP = MLP(input_dim = 784, n_nodes_hl = (500,500,500))
MLP.train(data)
MLP.test(data)
MLP.get_parameters()


### Convolutional Neural Network ###

ConvNet = Convolutional_Net(input_dim = 784, n_nodes_hl = (32, 64, 1024))
ConvNet.train(data)
ConvNet.test(data)
ConvNet.get_parameters()


### RNN_LSTM ###

lstm = RNN_LSTM(input_dim = 28, seq_size = 28)
lstm.train(data)
lstm.test(data)
lstm.get_parameters()
```

Figure 3.14: In this file we create an instance of each class. Thus we train and analyize them making use of their methods.

3.6 Evaluate the models

Once we have concluded the MLP, CNN and RNN implementation, we can train and test these algorithms identifying which achieves the best accuracy.
Moreover,thanks to OOP, it is possible to evaluate the previous models in a very clear way. Indeed we can define a `main.py` file where we create an instance of each of the three classes that can be trained and evaluated simply evoking the right method (see Fig.3.14).
If the training of both MLP and RNN with LSTM cell results to be relatively fast, in the case of CNN we have been waiting more than an hour to see the training phase completed. This is due to the fact that implementing a CNN requires a lot of computational power and doing it with a CPU needs a lot of time.
The using of a *graphics processing unit* (GPU) helps us to experiment in deep learning field in a more efficient way. GPUs are very efficient at manipulating computer graphics and image processing and their parallel structure makes them more efficient than CPUs for algorithms where the processing of a large block of data is done in parallel.

```
Python 3.5.2+ (default, Sep 22 2016, 12:18:14)
[GCC 6.2.0 20160927] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
====== RESTART: /home/gabriele/MyTensorFlowStuff/Tesi/Chapter_3/main.py ======
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz

                           Multilayer Perceptron

Accuracy 0.962

                             RNN with LSTM cell

Accuracy 0.9807
LSTM:
 weights:
 [[-0.45044929  0.59757727  0.17531581 ..., -0.17109507 -2.41479349
  -0.29153055]
 [-0.02775256  0.11609401 -0.9888714  ..., -0.61103594 -0.15647443
   0.7813012 ]
 [ 0.0769309   1.18899691  0.47634664 ...,  0.15962665  0.25788802
   1.73599696]
 ...,
 [-0.4480013  -0.39525989  0.04642415 ..., -0.06807691 -0.59643549
   0.36603001]
 [ 0.02432841  0.01332261  0.72658455 ..., -0.05710357 -1.61783648
  -0.30994287]
 [ 0.56055468 -1.71535039 -0.46924511 ...,  0.13135456  0.54054499
  -0.16457391]]
 biases:
 [-1.17694783  0.33383852  1.38335598 -0.69956762  0.39959899  0.84649724
 -1.08694625 -3.22479534 -0.93607819 -1.22476721]

                         Convolutional Neural Network

Accuracy 0.974
```

Figure 3.15: The performaces of the three algorithms are shown in the picture. The RNN with LSTM cell achieves the best performance. Moreover, the RNN parameters are displayed. Those of CNN and MLP are omitted in order to make the visualization comprehensible.

The accuracies achieved are shown in Fig.3.15 . As it is possible to see, the implemented RNN with LSTM cell achieves the best performance with only 5 epochs. Furthermore, by finding the optimal number of epochs and adding drop out we could augment the accuracy of the model. However, making the CNN more powerful we can exceed the RNN performance. Indeed the actual state of the art is achieved using a convolutional architecture that makes an error rate of 0.21% (Wan et al., 2013; 16).

Chapter 4

A machine to forecast financial time-series

In this chapter, we are going to describe the construction of the object of the thesis: a machine capable to predict future realization of a financial time-series. The key idea is to develop an architecture able to forecast future values of a quantity interesting for us, starting from a financial scenario rather than the only previous realizations of this quantity. We will do it by combining two different deep learning models: the *deep autoencoder* and the *RNN* with *LSTM cell*. Therefore, unlike the methods so far analysed, this machine extends the concept of *depth*: it is a deep architecture composed by two consecutive parts each representing a deep model.
The first, i.e. the deep autoencoder, elaborates the financial scenario at the step t squashing the realizations $x_{i,t}$ of i different financial time-series. The second, i.e. the LSTM, makes predictions using the squashed representation given by the deep autoencoder. In fig. 4.1 , the complete model is shown.
Before treating the two elements of the machine, we are going to examine in depth the optimizer we will apply: *Adam*.

4.1 Adaptive moments (ADAM)

Widely speaking optimization algorithms can be mainly divided in two classes:

- *batch* (or *deterministic*) gradient methods: they process all of the training examples simultaneously in a large batch. This terminology can be somewhat confusing because the word "batch" often refers to the *minibatch* used by stochastic gradient methods. Typically the term "batch gradient descent" implies the use of the full training set, while the use of the term "batch" to describe a group of examples does not. For example, it is very common to use the term "batch size" to describe the size of a minibatch.

- *stochastic* (or *online*) gradient methods: they process a group of examples exploiting minibatches, i.e. using one or more than one instances but less than all of the training examples.

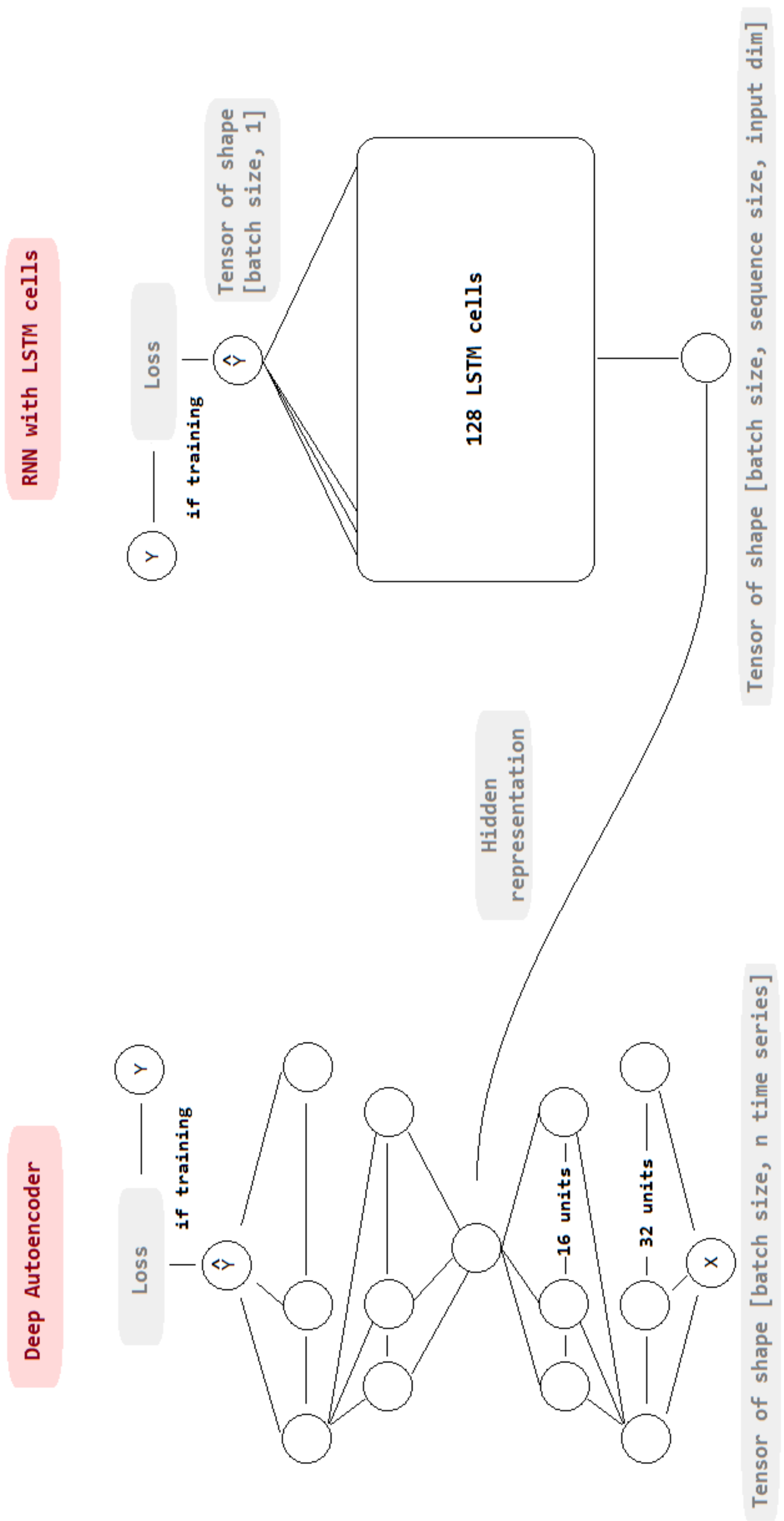


Figure 4.1: This figure shows the structure of the proposed machine. By starting from a financial scenario (`hidden representation`) elaborated by the deep autoencoder whose input is a finite number of financial time-series, the model forecasts the future realization of a financial quantity interesting for us by using a recurrent neural network with LSTM cells.

Stochastic gradient-based optimization is of core practical importance in many fields of science and engineering. Many problems in these fields can be cast as the optimization of some scalar parametrized objective function requiring maximization or minimization with respect to its parameters.
If the function is differentiable w.r.t. its parameters, gradient descent is a relatively efficient optimization method, since the computation of first-order partial derivatives w.r.t. all the parameters is of the same computational complexity as just evaluating the function.
Often, objective functions are stochastic. For example, many objective functions are composed of a sum of sub-functions evaluated at different sub-samples of data; in this case optimization can be made more efficient by taking gradient steps w.r.t. individual sub-functions, i.e. *stochastic gradient descent* (SGD) or *ascent*.
SGD proved itself as an efficient and effective optimization method that was central in many machine learning success stories, such as recent advances in deep learning. Objectives may also have other sources of noise than data sub-sampling, such as drop out regularization. For all such noisy objectives, efficient stochastic optimization techniques are required. *Adam* is a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The name Adam is derived from *adaptive moment estimation*. The method is designed to combine the advantages of two popular methods: *AdaGrad* (Adaptive Gradient; Duchi et al. 4), which works well with sparse gradients, and *RMSProp* (Root Mean Square Propagation; Hinton 8), which works well in on-line and non-stationary settings. Both are *adaptive algorithms*.
The main advantage of this kind of optimizers is that they avoid the difficult choice of the learning rate, an hyperparameter that has significant impact on the performance of the model. Indeed, this class of algorithms individually adapts the learning rates of all model parameters using a particular rule that takes care of some quantities associated to the gradient. For instance, Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.
Some of Adam's advantages are that the magnitudes of parameter updates are invariant to rescaling of the gradient, its step sizes are approximately bounded by the step size hyperparameter, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of step size annealing.
The algorithm can be described as follows. Let $f(\theta)$ be a noisy objective function: a stochastic scalar function that is differentiable w.r.t. parameters vector θ. We are interested in minimizing the expected value of this function, $\mathbf{E}|f(\theta)|$ w.r.t. its parameters θs. Let us denote with $\{f_1(\theta), ..., f_T(\theta)\}$ the realizations of the stochastic function at subsequent time steps $\{1, ..., T\}$. The stochasticity might come from the evaluation at random sub-samples (minibatches) of data points, or arises from inherent function noise. With $g_t = \nabla_\theta f_t(\theta)$ we denote the gradient, i.e. the vector of partial derivatives of f_t, w.r.t θ evaluated at time step t.
The algorithm updates exponential moving averages of the gradient m_t and the squared gradient v_t where the hyperparameters $\beta_1,\ \beta_2 \in [0; 1)$ control the exponential decay rates of these moving averages. The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the uncen-

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 4.2: Adam algorithm. (From 11)

tered variance) of the gradient. However, these moving averages are initialized as vectors of zeros, leading to moment estimates that are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. the βs are close to 1). The good news is that this initialization bias can be easily counteracted, resulting in bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$.
The algorithm is shown in Fig. 4.2. Good default settings are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

4.2 Deep autoencoder

Before introducing a *deep autoencoder*, it is well worth talk about its predecessor: the *autoencoder*. Suppose we have a set of unlabelled training examples $x_1, x_2, \ldots, x_n$, where $x_i \in \Re^n$. An autoencoder neural network is an unsupervised learning algorithm which sets the target values to be equal to the inputs. In other words, it uses $y_i = x_i$. The autoencoder tries to learn an approximation to the identity function, so that the output y is similar to x. The network may be viewed as consisting of two parts: an encoder function $h = f(x)$ and a decoder that produces a reconstruction $y = g(h)$ (see Fig. 4.3).
The identity function seems a particularly trivial and not interesting function in terms of learning, but by placing constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data. Indeed, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data. Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings $p_{encoder}(h|x)$ and $p_{decoder}(x|h)$. The idea of autoencoders has been part of the historical landscape of neural networks for decades. Traditionally, autoencoders were used for dimension-

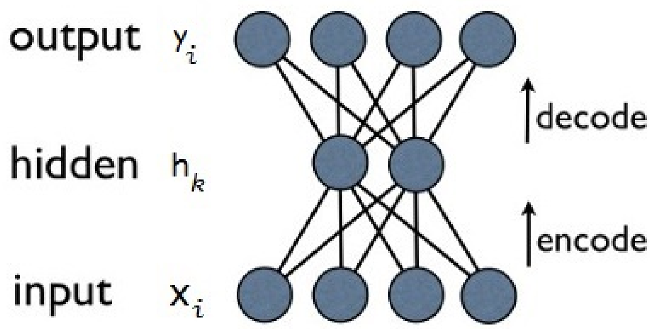


Figure 4.3: The general structure of an autoencoder, it maps an input x to an output(called reconstruction)y through an internal representation or code h. (From www.quora.com)

ality reduction or feature learning.

Autoencoders may be thought of as being a special case of feedforward networks, and may be trained with all of the same techniques, typically minibatch gradient descent following gradients computed by back-propagation. For completeness, it is fair to say that autoencoders may also be trained using *recirculation*, a learning algorithm based on comparing the activations of the network on the original input to the activations on the reconstructed input. Recirculation is regarded as more biologically plausible than back-propagation, but is rarely used for machine learning applications.

Moreover, copying the input to the output may sound useless, but we are typically not interested in the output of the decoder. Instead, we hope that training the autoencoder to perform the input copying task will result in h taking on useful properties. One way to obtain useful features from the autoencoder is to constrain h to have smaller dimension than x. An autoencoder whose code dimension is less than the input dimension is called *undercomplete*. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data. The learning process is simply described as minimizing a loss function $L(x, g(f(x)))$, where L is a loss function penalizing $g(f(x))$ for being dissimilar from x, such as the mean squared error. In addition, an autoencoder whose hidden code is allowed to have dimension greater than the input is said *overcomplete*.

Similarly to its predecessor, a deep autoencoder is composed of two, symmetrical deep-belief networks that typically have more than one hidden layers: the first representing the encoding half of the network and the second describing the decoding part (see Fig. 4.4).

The multilayer structure of the deep autoencoder might cause benefits in terms of computational cost and compression quality. Indeed, the *Universal Approximation Theorem* guarantees that a feedforward neural network with at least one hidden layer can represent an approximation of whatever function with an arbitrary accuracy degree as long as it has enough hidden units. Therefore, a greater number of hidden layers guarantees the same result and permits to have a lower number of hidden units per layer.

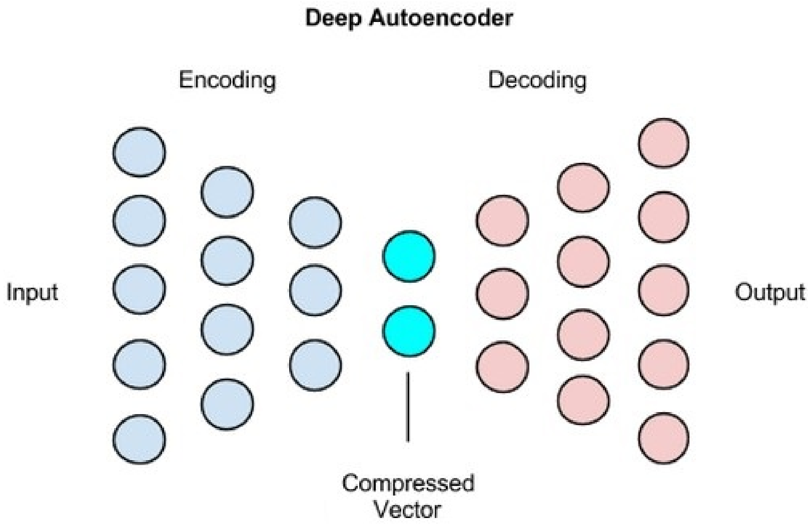


Figure 4.4: The general structure of a deep autoencoder, mapping an input x to an output$y$. This mapping is realized making use of more than one hidden layer. Each layer is fully-connected to the next. The symmetrical shape of the model is commonly know as "bottleneck structure". (From `deeplearning4j.org`)

Once having introduced the deep autoencoder we can focus on its implementation(see Fig. 4.5 and Fig. 4.6).
As in the previous chapter, we are going to use the convenient OOP, thus the model will represent a `class` . The deep autoencoder is composed by an encoder and a decoder both consisting of 3 layers. Since the training will be done using mini-batch, the temporal realizations of the time-series are stored in the tensor `X` having shape *(batch size, number of time-series)*. Thus, given a finite number of financial time-series, our goal is to get the final compressed representation, given back by the third encoder layer. Indeed, this compressed result can be interpreted as a financial scenario and it is what we will give as input to the RNN with LSTM cell.
The operation involved in building the encoder and the decoder are completely described in Fig. 4.5 and Fig. 4.6. Before passing to the loss function and optimizer analysis, please notice that we copy both `output_layer_3_pass` and `output_layer_6` to `self.encoded` and `self.decoded` respectively. These operations are fundamental: the first allows us to store our target, the compressed representation; the second makes possible to define the cost function and the accuracy of our model.
As shown in Fig. 4.6 , we choose `tf.nn.l2_loss` as loss function , defined by the formula:

$$Loss = \frac{1}{2}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{4.1}$$

where $\hat{Y}$ is the predictions' vector and Y the targets' vector. Once again the optimization of the cost function is made using the optimizer described previously, *Adam*.

```
This file creates a deep autoencoder. It is composed by an encoder and a decoder both consisting of
3 layers.
The values of the time series are stored in the tensor X, that has shape 'batch size'
by 'number of time series'.
Given a finite number of financial time series, the goal is to get the final compressed representation
given back by the third encoder layer. This compressed result can be interpreted as a
financial scenario.
Hyperparameters:
 - input_dim: vector containing the values of the time series at each temporal realization;
 - epochs: how long the deep autoencoder is trained;
 - batch_size: 128. This number is chosen in honor of those familiar numbers 32, 64, 128, 256... ;
 - learning_rate: it allows to change the default learning rate of the optimizer;
 - n_examples: number of instances to visualize the input, compressed and decoded values.
'''

import tensorflow as tf
import numpy as np

class Deep_Autoencoder:
    def __init__(self, input_dim, n_nodes_hl = (32,16,1), epochs = 400, batch_size = 128, learning_rate = 0.02, n_examples = 10):

        # Hyperparameters

        self.input_dim = input_dim
        self.epochs = epochs
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.n_examples = n_examples

        # Input and target placeholders

        X = tf.placeholder("float", [None, self.input_dim])
        Y = tf.placeholder("float", [None, self.input_dim])

        '''
        Define the bottleneck structure. Weights and biases of the layers are formalized as dictionaries.
        The weights of a layer represent a tensor of shape 'number of units of
        the previous layer' by 'number of units of the layer'.
        The biases of a layer represent a tensor of shape 'number of units of the layer'.
        The command tf.random_normal fill the tensors with random normal values.
        The sigmoid activation function is applied to the output of
        every layer except the last decoder layer.
        '''

        with tf.name_scope('encoder'):
            weights_e = {
                'encoder_h1' : tf.Variable(tf.random_normal([input_dim , n_nodes_hl[0]]), name = 'weights_e1'),
                'encoder_h2' : tf.Variable(tf.random_normal([n_nodes_hl[0], n_nodes_hl[1]]), name = 'weights_e2'),
                'encoder_h3' : tf.Variable(tf.random_normal([n_nodes_hl[1], n_nodes_hl[2]]), name = 'weights_e3')
                }

            biases_e = {
                'b_e1' : tf.Variable(tf.random_normal([n_nodes_hl[0]]), name = 'biases_e1'),
                'b_e2' : tf.Variable(tf.random_normal([n_nodes_hl[1]]), name = 'biases_e2'),
                'b_e3' : tf.Variable(tf.random_normal([n_nodes_hl[2]]), name = 'biases_e3')
                }

            output_layer_1 = tf.add(tf.matmul(X, weights_e['encoder_h1']), biases_e['b_e1'])
            output_layer_1 = tf.nn.sigmoid(output_layer_1)

            output_layer_2 = tf.add(tf.matmul(output_layer_1, weights_e['encoder_h2']), biases_e['b_e2'])
            output_layer_2 = tf.nn.sigmoid(output_layer_2)

            output_layer_3_pass = tf.add(tf.matmul(output_layer_2, weights_e['encoder_h3']), biases_e['b_e3'])
            output_layer_3 = tf.nn.sigmoid(output_layer_3_pass)
```

Figure 4.5: Deep autoencoder `class`: `__init__` (part 1)

```
with tf.name_scope('decoder'):
    weights_d = {
        'decoder_h1' : tf.Variable(tf.random_normal([n_nodes_hl[2], n_nodes_hl[1]]), name = 'weights_d1'),
        'decoder_h2' : tf.Variable(tf.random_normal([n_nodes_hl[1], n_nodes_hl[0]]), name = 'weights_d2'),
        'decoder_h3' : tf.Variable(tf.random_normal([n_nodes_hl[0], input_dim]), name = 'weights_d3')
        }

    biases_d = {
        'b_d1' : tf.Variable(tf.random_normal([n_nodes_hl[1]]), name = 'biases_d1'),
        'b_d2' : tf.Variable(tf.random_normal([n_nodes_hl[0]]), name = 'biases_d2'),
        'b_d3' : tf.Variable(tf.random_normal([input_dim]), name = 'biases_d3')
        }

    output_layer_4 = tf.add(tf.matmul(output_layer_3, weights_d['decoder_h1']), biases_d['b_d1'])
    output_layer_4 = tf.nn.sigmoid(output_layer_4)

    output_layer_5 = tf.add(tf.matmul(output_layer_4, weights_d['decoder_h2']), biases_d['b_d2'])
    output_layer_5 = tf.nn.sigmoid(output_layer_5)

    output_layer_6 = tf.add(tf.matmul(output_layer_5, weights_d['decoder_h3']), biases_d['b_d3'])

self.X = X
self.Y = Y
self.weights = { 'weights_e' : weights_e, 'weights_d' : weights_d}
self.biases = { 'biases_e' : biases_e, 'biases_d' : biases_d}
self.encoded = output_layer_3_pass
self.decoded = output_layer_6

# Loss function and optimizer

self.cost = tf.nn.l2_loss(self.decoded - self.Y)
self.optimizer = tf.train.AdamOptimizer(self.learning_rate).minimize(self.cost)

# The args are normalized to the targets

self.accuracy = 1. - (tf.reduce_mean(tf.abs((self.decoded - self.Y)/self.Y )))

# Needed to save the session

self.saver = tf.train.Saver()
```

Figure 4.6: Deep autoencoder `class`: `__init__` (part 2)

Finally, `self.accuracy` gives us a measure of how good is our model. In particular, we evaluate the deep autoencoder looking at the reconstruction capability of it: we compute the absolute error (normalized to the targets' vector) and thus we take the mean. The normalization ensures equally weighted absolute error for every time-series realization, in this way we avoid that time-series with very low values overestimate the accuracy of the model.
At this point we define the saver object and then the usual `train`, `test` and `get_parameters` methods (see Fig. 4.7 , Fig. 4.8 and Fig. 4.9).
However, unlike the three examples of the previous chapter, both `train` and `test` methods will return `numpy arrays`: they will be the inputs of the LSTM training and testing phase, computed thanks to the definition of `self.encoded`.

```python
def train_neural_network(self, data, targets):
    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())

        for epoch in range(self.epochs):

            epoch_loss = 0
            i = 0

            # It will be the input of the LSTM

            hidden_vec = np.array([])

            # Training in batch mode

            while i < len(data):
                start = i
                end = i + self.batch_size

                batch_x = np.array(data[start:end])
                batch_y = np.array(targets[start:end])

                hidden, _, c = sess.run([self.encoded, self.optimizer, self.cost], feed_dict={ self.X: batch_x, self.Y: batch_y})
                epoch_loss += c
                hidden_vec = np.append(hidden_vec, hidden)
                i += self.batch_size

            if (epoch+1) % 50 == 0:
                print('Epoch', epoch + 1, 'completed out of', self.epochs, ', train loss:', epoch_loss)
                self.saver.save(sess, './Deep_Autoencoder.ckpt')

        self.saver.save(sess, './Deep_Autoencoder.ckpt')
        print('Accuracy', self.accuracy.eval({self.X: data, self.Y: targets}))
    return hidden_vec
```

Figure 4.7: Deep autoencoder `class`: `train` method

```python
def test_neural_network(self, data, targets):
    with tf.Session() as sess:

        # Invoke the saved session

        self.saver.restore(sess, './Deep_Autoencoder.ckpt')

        hidden_vec = np.array([])

        i = 0

        # Testing in batch mode

        while i < len(data):
                start = i
                end = i + self.batch_size

                batch_x = np.array(data[start:end])

                hidden = sess.run(self.encoded, feed_dict={self.X: batch_x})

                hidden_vec = np.append(hidden_vec, hidden)
                i += self.batch_size

        reconstruction = sess.run(self.decoded, feed_dict={self.X: data})

        # Show some results

        for i in range(self.n_examples):
            print('input: ', data[i], ', compressed: ', hidden_vec[i], ', output: ', reconstruction[i])

        # Accuracy evaluated on the whole test set

        print('Accuracy', self.accuracy.eval({self.X: data, self.Y: targets}))

    return hidden_vec
```

Figure 4.8: Deep autoencoder `class`: `test` method

```
def get_parameters(self):
    with tf.Session() as sess:
        self.saver.restore(sess, './Deep_Autoencoder.ckpt')
        weights, biases = sess.run([self.weights, self.biases])
        print('ENCODER: \n',
              'HIDDEN LAYER 1: \n',
              'weights: \n', weights['weights_e']['encoder_h1'], '\n',
              'biases: \n', biases['biases_e']['b_e1'], '\n',
              'HIDDEN LAYER 2: \n',
              'weights: \n', weights['weights_e']['encoder_h2'], '\n',
              'biases: \n', biases['biases_e']['b_e1'], '\n',
              'HIDDEN LAYER 3: \n',
              'weights: \n', weights['weights_e']['encoder_h3'], '\n',
              'biases: \n', biases['biases_e']['b_e1'], '\n'
              'DECODER: \n',
              'HIDDEN LAYER 1: \n',
              'weights: \n', weights['weights_d']['decoder_h1'], '\n',
              'biases: \n', biases['biases_d']['b_d1'], '\n',
              'HIDDEN LAYER 2: \n',
              'weights: \n', weights['weights_d']['decoder_h2'], '\n',
              'biases: \n', biases['biases_d']['b_d1'], '\n',
              'HIDDEN LAYER 3: \n',
              'weights: \n', weights['weights_d']['decoder_h3'], '\n',
              'biases: \n', biases['biases_d']['b_d1'], '\n'
              )
    return weights, biases
```

Figure 4.9: Deep autoencoder `class`: `get_parameters` method

4.3 RNNs: the long-term dependencies problem

As aforementioned in Chapter 3, even though RNNs represent a powerful class of model, they have a problem related to the learning of long-term dependencies. Indeed as already said, the basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode(rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories,with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given by long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones.
Recurrent networks involve the composition of the same function multiple times, once per time step. These compositions can result in extremely nonlinear behaviour. In particular, the function composition employed by recurrent neural networks somewhat resembles matrix multiplication. We can think of the recurrence relation

$$h_t = W^T h_{t-1}, \tag{4.2}$$

as a very simple recurrent neural network lacking a nonlinear activation function, and lacking inputs x. This recurrence relation essentially describes the power method. It may be simplified to

$$h_t = (W^T)^t h_0, \tag{4.3}$$

and if W admits an eigendecomposition of the form

$$W = Q\Lambda^t Q^T, \tag{4.4}$$

with orthogonal Q, the recurrence may be simplified further to

$$h_t = Q^T \Lambda^t Q h_0. \tag{4.5}$$

The eigenvalues are raised to the power of t causing the decaying toward zero of eigenvalues with magnitude less than one and the explosion of eigenvalues with magnitude greater than one. Any component of h_0 that is not aligned with the largest eigenvector will eventually be discarded. This problem is particular important in recurrent networks. For instance, in the scalar case, imagine multiplying a weight w by itself many times. The product w^t will either vanish or explode depending on the magnitude of w. The vanishing and exploding gradient problem for RNNs was independently discovered by separate researchers: Hochreiter (9) and Bengio et al. (2 , 3). One may hope that the problem can be simply avoided by staying in a region of parameter space where the gradients do not vanish or explode. Unfortunately, in order to store memories in a way that is robust to small perturbations, the RNN must enter a region of parameter space where gradients vanish.
Specifically, whenever the model is able to represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction. However, it does not mean that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. In practice, the experiments in Bengio et al. (3) show that as we increase the span of the dependencies that need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful training of a traditional RNN via SGD rapidly reaching 0 for sequences of only length 10 or 20. Several approaches have been proposed to reduce the difficulty of learning long-term dependencies. In particular, one of them is widely applied: the *long short-term memory* (LSTM). Since it represents the second part of the machine, it is well worth to focus on the development of this powerful model.

4.4 Long short-term memory cell (LSTM)

The LSTM model was developed by Hochreiter and Schimdhuber in 1997 (10). In its first form, the model was composed by a certain number of basic units, the *memory blocks*, each containing one or more *memory cells* and a pair of gating units which control inputs and outputs to all cells in the block. Every memory cell has at its core a recurrently self-connected linear unit called "*Constant Error Carousel*" (CEC), whose activation is known as *state*. The CEC solves the vanishing error problem: in the absence of new input or error signals to the cell, the CEC local error back flow remains constant , neither growing nor decaying. Moreover, the two gates protect the CEC from both forward flowing activation and backward flowing error. Indeed, when gates are closed (activation around zero), irrelevant inputs and noise do not enter the cell, and the cell state does not perturb the remainder of the network.
However, Gers et al. (5) in 1999 show that even the first LSTM model fails to learn to correctly process certain very long or continual time-series. Essentially, the problem was that a continual input stream eventually may cause the internal values of the cells to grow without bound. Therefore they overcame this problem adding

another gate: the *forget gate*. Fig. 4.10 shows this modified model.
The forget gate is designed to learn to reset memory blocks once their contents are useless for the model.
By considering to have an LSTM as that shown in Fig. 4.10 and to deal with discrete temporal intervals $t = 1, 2, ...$, we can compute the output of the LSTM cell as follows.
At the beginning, the input to the cell v belonging to the block j, $net_{c_j^v}(t)$, is squashed by a centered sigmoid g:

$$g(net_{c_j^v}) = g(\sum_m w_{c_j^v m} y^m(t-1)), \tag{4.6}$$

where $w_{c_j^v m}$ is the weight of the unit m to the cell c_j^v. At this point, the result is multiplied by the output $y^{in_j}(t)$, equal to:

$$y^{in_j}(t) = f_{in_j}(net_{in_j}(t)) = f(\sum_m w_{in_j m} y^m(t-1)), \tag{4.7}$$

with f sigmoid such that $y^{in_j}(t) \in [0, 1]$. In this way the inputs with $y^{in_j}(t)$ nearly zero, will be discarded.
In order to compute the state $s_{c_j^v}(t)$, it is now necessary to add to the previous product $g \cdot y^{in_j}$ the one between $s_{c_j^v}(t-1)$ and the forget gate's output y^{φ_j}, given by:

$$y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)) = f(\sum_m w_{\varphi_j m} y^m(t-1)), \tag{4.8}$$

$$s_{c_j^v}(t) = s_{c_j^v}(t-1) y^{\varphi_j}(t) + g(net_{c_j^v}(t)) y^{in_j}(t), \tag{4.9}$$

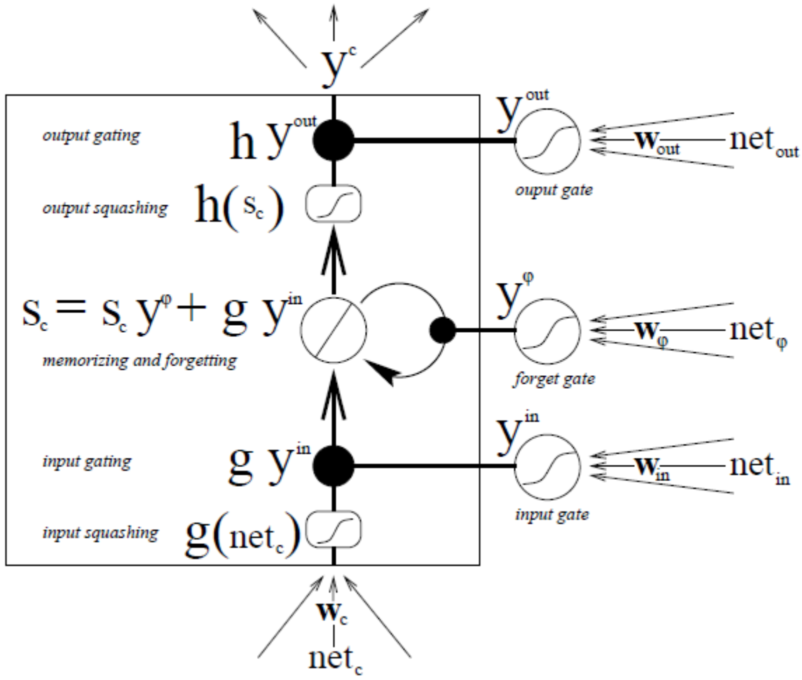
with $s_{c_j^v}(0) = 0$.
Therefore, we can compute the output of the cell $y^{c_j^v}(t)$ applying a centered sigmoid to $s_{c_j^v}(t)$ and multiplying this value by the output gate's result:

$$y^{c_j^v}(t) = y^{out_j}(t) h(s_{c_j^v}(t)) = f_{out_j}(net_{out_j}(t)) h(s_{c_j^v}(t)), \tag{4.10}$$

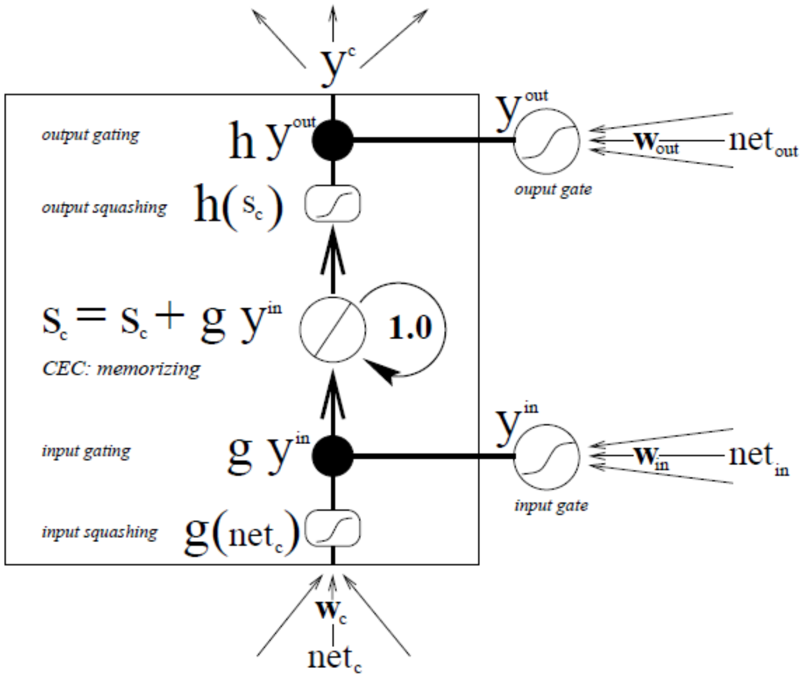
with:

$$f_{out_j}(net_{out_j}(t)) = f(\sum_m w_{out_j m} y^m(t-1)). \tag{4.11}$$

As shown in the RNN example of previous chapter, `TensorFlow` provides built-in classes and methods describing RNN algorithms. In the following implementation we will use the `class tf.nn.rnn_cell.LSTMCell`, based on Sak et al., 2014. The main contribute of this paper was to introduce a two-layer deep LSTM where each LSTM layer has a linear recurrent projection layer connected to the input of the LSTM. This model is also known as LSTMP. They showed that this architecture converges quickly and could exceed state-of-art in speech recognition performance. Moreover, by using the `class tf.nn.rnn_cell.LSTMCell` we implement a slightly modified version of the one proposed by Gers. et al, previously analysed. As shown in Fig. 4.11 , the architecture is pretty the same. The improvement consists in



a) Modified LSTM



b) Standard LSTM

Figure 4.10: Modified version of the LSTM. The figure shows a memory block with only one cell. In general each block can contain more cells. In addition the constant 1, i.e. the weight of the standard LSTM CEC (part b) is replaced by the multiplicative forget gate activation y^{φ}.

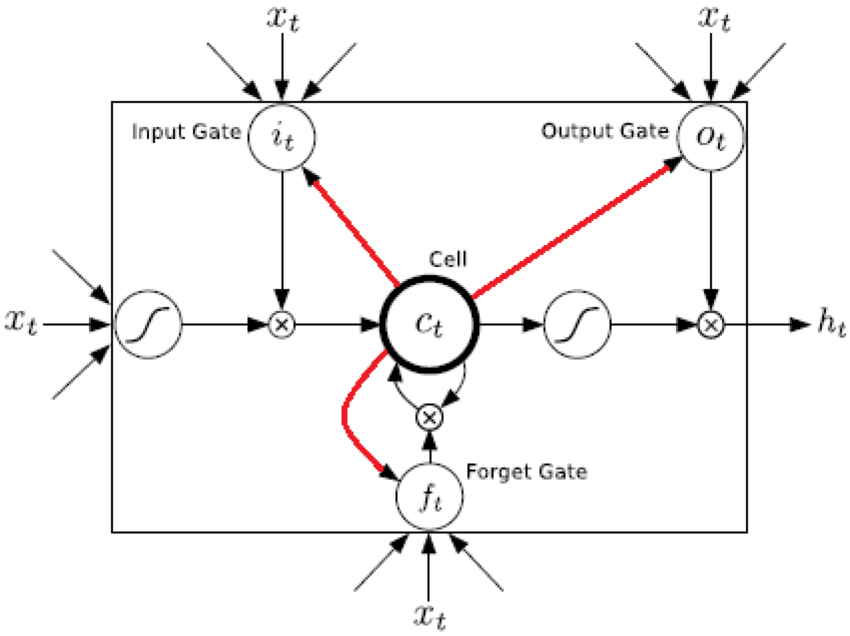


Figure 4.11: LSTM architecture used in the second part of the machine. In red the "peephole connections". (From 7)

the introduction of additional "peephole connections" from the CEC to the gates, as proposed by Gers. et al in 2002 (6). The model is described by the following equations:

$$\begin{cases} i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \\ f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\ c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \\ h_t = o_t tanh(c_t) \end{cases} \tag{4.12}$$

where σ is the logistic sigmoid function, and i, f, o and c are respectively the *input gate, forget gate, output gate, cell* and *cell input* activation vectors, all of which are the same size as the hidden vector h. The weight matrix subscripts have the usual meaning, for example W_{hi} is the hidden-input gate matrix. The weight matrices from the CEC to gate vectors are diagonal, so element m in each gate vector only receives input from element m of the cell vector.
By having analysed from a theoretical point of view the LSTM, we can finally discuss the implementation of the second part of the machine (Fig. 4.12). As it is possible to notice, the code is very similar to that shown in Fig. 3.11 and Fig. 3.13 . This is one of the key aspect of deep learning: the amazing flexibility of the models, i.e. it does not take much to make suitable a model to our task. Therefore we will focus on the introduced changes.
The first modification we find in the code is the loss function: in this implementation we adopt the mean squared error as in the deep autoencoder part. Once again the optimization is assigned to the Adam method and the model is evaluated in terms of its reconstruction capability.

```
'''
This part is relative to the creation of the RNN with LSTM cell. The data are
stored in a tensor X of shape 'batch size' by 'sequence size' by 'input dimension'.
Hyperparameters:
 - input_dim: dimension of the input;
 - seq_size: number of sequences;
 - hidden_dim: RNN size;
 - epochs: how long the RNN is trained;
 - batch_size: 128. This number is chosen to cite those familiar numbers 32, 64, 128, 256... ;
 - learning_rate: it allows to change the default learning rate of the optimizer;
 - keep_prob: it makes possible an optional drop out one of the most common
   regularization tecnique. In this case it is set to 1.0 (consequently the
   drop out is not applied).
'''

import tensorflow as tf
import numpy as np
from tensorflow.python.ops import rnn, rnn_cell
import matplotlib.pyplot as plt

class RNN_LSTM:

    def __init__(self, input_dim, seq_size, hidden_dim = 128, learning_rate = 0.01, batch_size = 128, epochs = 200, keep_prob = 1.0):

        # Hyperparameters

        self.input_dim = input_dim
        self.seq_size = seq_size
        self.hidden_dim = hidden_dim
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.epochs = epochs
        self.keep_prob = keep_prob

        # Input and target placeholders

        X = tf.placeholder("float", [None, self.seq_size, self.input_dim])
        Y = tf.placeholder("float", [None, 1])

        # Weights and biases

        weights = tf.Variable(tf.random_normal([self.hidden_dim, 1]), name='weights')
        biases = tf.Variable(tf.random_normal([1]), name='b')

        self.X = X
        self.Y = Y
        self.weights = weights
        self.biases = biases
        self.output = self.lstm()

        # Cost optimizer

        self.cost = tf.nn.l2_loss(self.output - self.Y)
        self.optimizer = tf.train.AdamOptimizer(self.learning_rate).minimize(self.cost)

        self.accuracy = 1. - (tf.reduce_mean(tf.abs((self.output - self.Y)/self.Y )))

        # Needed to save the session

        self.saver = tf.train.Saver()

    def lstm(self):
        """
        Prepare the input shape for the lstm. Actual shape is (batch_size, seq_size, input_dim),
        the required shape is 'seq_size' tensors list of shape (batch_size, n_input)
        """
        X = self.X

        # Permuting batch_size and seq_size
        X = tf.transpose(X, [1, 0, 2])
        # Reshaping to (seq_size*batch_size, input_dim)
        X = tf.reshape(X, [-1, self.input_dim])
        # Split to get a list of 'seq_size' tensors of shape (batch_size, input_dim)
        X = tf.split(0, self.seq_size, X)

        # Create lstm and add the dropout to each output of a cell

        lstm_cell = rnn_cell.LSTMCell(self.hidden_dim, use_peepholes=True)
        lstm_cell = rnn_cell.DropoutWrapper(lstm_cell, input_keep_prob=self.keep_prob, output_keep_prob=self.keep_prob)
        outputs, states = rnn.rnn(lstm_cell, X, dtype=tf.float32)

        output = tf.matmul(outputs[-1], self.weights) + self.biases

        return output
```

Figure 4.12: RNN `class`: `__init__`

```
def train(self, data, target):

    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())

        for epoch in range(self.epochs):
            epoch_loss = 0
            i = 0                                          Y = tf.placeholder("float", [None, 1])

            while i < int(len(data)/self.batch_size):

                start = i * self.batch_size
                end = (i+1) * self.batch_size

                batch_x = np.array(data[start:end])
                batch_y = np.array(target[start:end])

                batch_x = np.reshape(batch_x, (self.batch_size, self.seq_size, self.input_dim))
                batch_y = np.reshape(batch_y, (self.batch_size, 1))

                _, c = sess.run([self.optimizer, self.cost], feed_dict={self.X: batch_x, self.Y: batch_y})
                epoch_loss += c
                i += 1

            if (epoch+1) % 100 == 0:
                print('Epoch', epoch + 1, 'completed out of', self.epochs, ', train loss:', epoch_loss)

        save_path = self.saver.save(sess, './lstm.ckpt')
        print('Accuracy', self.accuracy.eval({self.X: data.reshape((-1, self.seq_size, self.input_dim)),
                                              self.Y: target.reshape(-1, 1)}))
def test(self, data, target):
    with tf.Session() as sess:

        self.saver.restore(sess, './lstm.ckpt')
        prediction = sess.run(self.output, feed_dict={self.X: data.reshape((-1, self.seq_size, self.input_dim))})
        target.reshape((-1, 1))

        for i in range(10):
            print('prediction: ', prediction[i], 'target: ', target[i])
        print('Accuracy', self.accuracy.eval({self.X: data.reshape((-1, self.seq_size, self.input_dim)),
                                              self.Y: target.reshape(-1, 1)}))
        plt.plot(prediction, linewidth=0.3, color='r', label='Predicted values')
        plt.plot(target, linewidth=0.3, color='g', label='Target values')
        plt.legend()
        plt.show()
```

Figure 4.13: RNN `class`: `train` and `test` method

Moreover, also the `train` and `test` are only slightly modified. Indeed a pair of adjustment have been made: to allow the training in minibatch mode and to make the `test` method showing the predictions. This last one operation is realized using `matplotlib.pyplot`, imported at the beginning of the document. It is a collection of command style functions that makes `matplotlib` (2D plotting libraries) working like MATLAB.
This conclude the building of the machine. In the next chapter we are going to see how to apply it to financial time-series and what results this approach gives us.

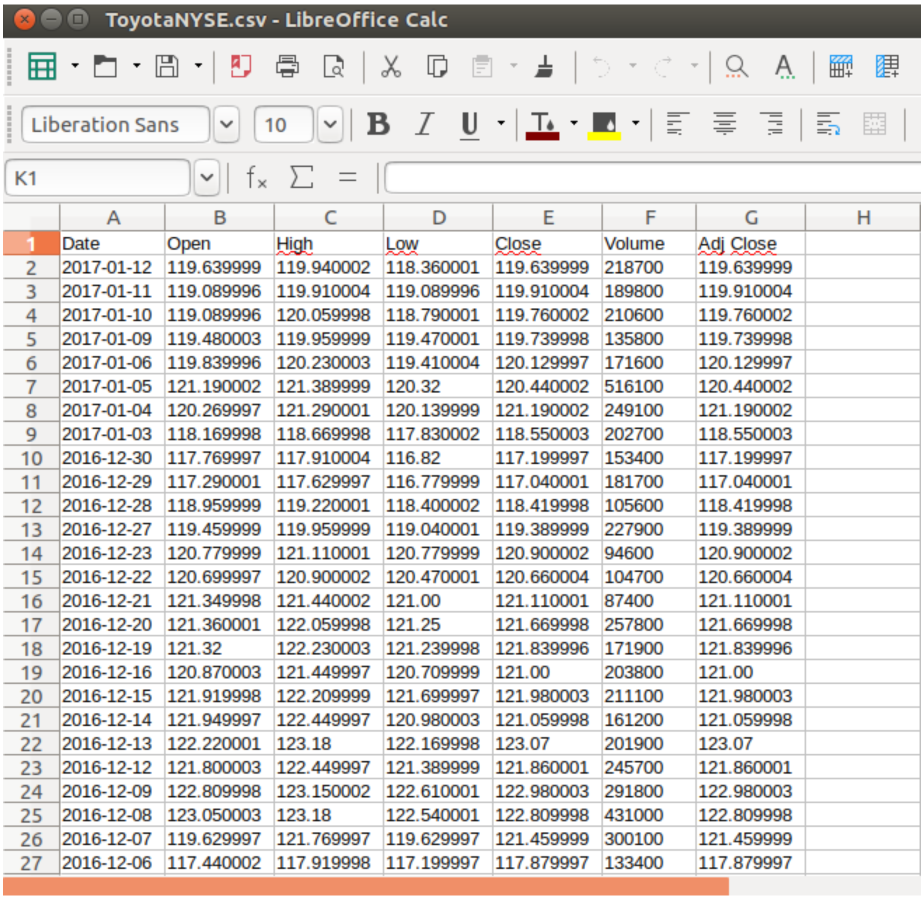
Chapter 5

Model's applications and analysis of results

In this chapter, the two previously shown algorithms are combined and subsequently applied to a financial dataset consisting of 10182 temporal realizations of **Toyota Motor Corporation** stock price. In particular, the data were downloaded from `finance.yahoo.com` and represent the Toyota stock values (TM) in daily format on the *New York Stock Exchange* (NYSE) from 18th August 1976 to 12th January 2017. NYSE is an American stock exchange located in New York and founded on May 17, 1792. It represents the world's largest stock exchange by market capitalization of its listed companies.
Someone may wonder about the choice of Toyota stock: we decided to analyse this time-series since it represents the history of an automotive colossus and at the same time it is long enough to satisfy the dataset size requirement necessary to deep learning models to achieve good performances. Indeed, one of the most important part when doing machine learning is to have a suitable dataset for the application. Fortunately, a lot of machine learning database composed by thousands of objects have been made available on-line. This accessibility was the main cause of the renaissance of AI field, as seen in the first chapter, indeed it is fundamental for the achievement of high performances of our machine learning algorithms.
As far as my researches are concerned, it was difficult to find an appropriate amount of available data for training and testing the machine: the time-series we would have liked to use alongside Toyota were not available in daily format and were composed by a too small number of temporal realizations. However, the results achieved are very interesting and made some considerations possible.
In the following sections we will see how to deal with our financial dataset and how to combine the deep autoencoder and the LSTM in the `main.py` file. Thus, we will analyse the results we got applying the method proposed and last but not least we will compare them to the ones achieved using only a RNN with LSTM cell.

5.1 Deep autoencoder feature sets

The data we deal with are daily Toyota stock prices on NYSE, as mentioned before. The file downloaded from `finance.yahoo.com` is a `.csv` (comma-separated values)



	A	B	C	D	E	F	G	H
1	Date	Open	High	Low	Close	Volume	Adj Close	
2	2017-01-12	119.639999	119.940002	118.360001	119.639999	218700	119.639999	
3	2017-01-11	119.089996	119.910004	119.089996	119.910004	189800	119.910004	
4	2017-01-10	119.089996	120.059998	118.790001	119.760002	210600	119.760002	
5	2017-01-09	119.480003	119.959999	119.470001	119.739998	135800	119.739998	
6	2017-01-06	119.839996	120.230003	119.410004	120.129997	171600	120.129997	
7	2017-01-05	121.190002	121.389999	120.32	120.440002	516100	120.440002	
8	2017-01-04	120.269997	121.290001	120.139999	121.190002	249100	121.190002	
9	2017-01-03	118.169998	118.669998	117.830002	118.550003	202700	118.550003	
10	2016-12-30	117.769997	117.910004	116.82	117.199997	153400	117.199997	
11	2016-12-29	117.290001	117.629997	116.779999	117.040001	181700	117.040001	
12	2016-12-28	118.959999	119.220001	118.400002	118.419998	105600	118.419998	
13	2016-12-27	119.459999	119.959999	119.040001	119.389999	227900	119.389999	
14	2016-12-23	120.779999	121.110001	120.779999	120.900002	94600	120.900002	
15	2016-12-22	120.699997	120.900002	120.470001	120.660004	104700	120.660004	
16	2016-12-21	121.349998	121.440002	121.00	121.110001	87400	121.110001	
17	2016-12-20	121.360001	122.059998	121.25	121.669998	257800	121.669998	
18	2016-12-19	121.32	122.230003	121.239998	121.839996	171900	121.839996	
19	2016-12-16	120.870003	121.449997	120.709999	121.00	203800	121.00	
20	2016-12-15	121.919998	122.209999	121.699997	121.980003	211100	121.980003	
21	2016-12-14	121.949997	122.449997	120.980003	121.059998	161200	121.059998	
22	2016-12-13	122.220001	123.18	122.169998	123.07	201900	123.07	
23	2016-12-12	121.800003	122.449997	121.389999	121.860001	245700	121.860001	
24	2016-12-09	122.809998	123.150002	122.610001	122.980003	291800	122.980003	
25	2016-12-08	123.050003	123.18	122.540001	122.809998	431000	122.809998	
26	2016-12-07	119.629997	121.769997	119.629997	121.459999	300100	121.459999	
27	2016-12-06	117.440002	117.919998	117.199997	117.879997	133400	117.879997	

Figure 5.1: The figure shows part of the `.csv` file downloaded from `finance.yahoo.com` relative to Toyota stock price on NYSE.

showing in its columns **date**, **opening price**, **high value**, **low value**, **closing price**, **volume exchanged** and **adjusted closing price**, i.e. stock closing price including all distributions and corporate actions that occurred at any time prior the following opening day.
As it is possible to see in Fig. 5.1 , the data go from the most recent to the oldest. Thus, the first necessary operation is to invert the dataset.
At this point we handle the dataset in a `featuresets.py` file to create the deep autoencoder training and testing sets.
As shown in Fig. 5.2 , we exploit the *csv Python module* for this operation. It implements classes to read and write tabular data in CSV format. The code consists of two main functions:

- `sample_handling(fname)` takes as input the `.csv` file and returns an array whose entries are `lists` representative of the rows of the file;

- `create_feature_sets(fname, test_size)` creates the train and test samples according to the value of `test_size`. This last one represents the percentage of instances gathered in the test set. It is well worth to notice how the dataset is split: the 80% is assigned to the training set and the 20% to the test set. The reason why is that to include the data relative to the crisis of

```python
'''
In this part the dataset is created. First of all, the data are stored in a
vector using the csv library of Python. Successively the dataset is created:
    - Train_x, Train_y are used to train the autoencoder. Trained on a shuffled
      features set the algorithm reaches higher accuracy;
    - train_x, train_y are the ordered version of Train_x and Train_y.
      The algorithm is tested on them. It is necessary to get ordered
      hidden_train vector for the LSTM implementation;
    - test_x, test_y respresent the test sample.
'''

import csv
import numpy as np
import random

def sample_handling(fname):
    featureset = []

    with open(fname, 'r') as csvfile:
        readCSV = csv.reader(csvfile, delimiter=',')

        feature = []
        for row in readCSV:
            feature = list(row[1:7])
            featureset.append(feature)

    return featureset

def create_feature_sets(fname, test_size = 0.2):

    features = []
    features = sample_handling(fname)

    features = np.array(features, dtype=np.float32)

    testing_size = int(test_size * len(features))

    Train_x = list(features[:-testing_size])
    random.shuffle(Train_x)
    Train_y = Train_x

    train_x = list(features[:-testing_size])
    train_y = train_x


    test_x = list(features[-testing_size:])
    test_y = list(features[-testing_size:])

    return Train_x, Train_y, train_x, train_y, test_x, test_y
```

Figure 5.2: The figure shows the `featursets.py` file.

2007/2008 yields to better performances of model.

Furthermore, it is important to focus on what happens to the second function. First of all, notice that the target values corresponds to the input ones: indeed as discussed in the previous chapter, the deep autoencoder tries to learn an approximation of the identity function. In addition, consider `Train_x` and `Train_y` variables, representing the shuffled version of the training set. This operation is applied since it has been noticed that it improves the performance of the deep autoencoder. Therefore, we train the algorithm using this sample. However, to forecast the time-series values, we need to apply the RNN with LSTM cell on an ordered dataset. This is the reason why we define `train_x` and `train_y` which are fundamental to get ordered hidden representation necessary to the LSTM. In particular, as it will be shown in the `main.py` file, we will perform this operation testing the deep autoencoder on both `train_x` and `test_x`.
Once built the deep autoencoder dataset we can focus on the construction of the LSTM features set.

5.2 LSTM feature set

At this point we have to define the LSTM dataset. Consider we applied the deep autoenconder on the previously described dataset.
As aforesaid, we would like to make forecasts on a financial scenario rather than on previous temporal realizations of the series, so we are interested in using the compressed representation of the deep autoencoder, i.e. the output of the layer 3 (see Fig 4.5). Thus the hidden vector will become the input of the LSTM and the target values will trivially be represented by the time-series we want to predict.
In Fig. 5.3 , we can see the `lstm_featuresets.py` file composed by a single function: `create_lstm_dataset`. This takes as arguments the train set, the test set and the `look_back`. In particular, `look_back` represents the sequence of data we pass to the LSTM to predict the next stock price value. For instance, consider the word "thesis" and suppose we would like to predict one of its character using the previous two: in this case our `look_back` will be 2.
Moreover it is fundamental to pay attention on how the train and test targets were stored in `train_Y` and `test_Y`. This operation has been realized appending to these vectors only the column values of the CSV file we are interested in.
At this point we are ready to perform the machine described in the previous chapter.

5.3 Applying the model to Toyota time-series

5.3.1 Forecasting Toyota closing values

Initially the model was evaluated in order to forecast Toyota daily closing values at different horizons. More precisely we tried to make a prediction $\hat{y}_{t+i}$ where $i \in \{1, 2, 3, 4, 5\}$ refers to negotiable days.
The first result we obtained was the benefit given by the use of the deep autoencoder as a sort of preprocessing on the single time-series of closing values. Indeed, as Fig.

```python
'''
In this part the lstm dataset is created. In particular, given the initial
autoencoder train and test sets, a new features set is generated taking in
account the look_back: in other words the sequence of data we use to predict
the value at the next time. For instance, consider the word 'thesis' stored
in the list [t, h, e, s, i, s] and suppose we want to use 2 character to
predict the next one (therefore look_back is 2). The dataset shape will be:

    train_x     train_y
    [t, h]  --> [e]
    [h, e]  --> [s]
    [e, s]  --> [i]
    and so on...

'''
import numpy as np

def create_lstm_dataset(train_x, train_y, test_x, test_y, look_back):

    train_X, train_Y, test_X, test_Y = [], [], [], []

    for i in range(len(train_x)-look_back-1):
        item_x = train_x[i:(i+look_back)]
        train_X.append(item_x)
        train_Y.append(train_y[i + look_back][0])

    for i in range(len(test_x)-look_back-1):
        item_x_t = test_x[i:(i+look_back)]
        test_X.append(item_x_t)
        test_Y.append(test_y[i+look_back][0])

    return np.array(train_X), np.array(train_Y), np.array(test_X), np.array(test_Y)
```

Figure 5.3: The figure shows the `lstm_featuresets.py` file.

5.4 displays it leads to higher performances w.r.t the case in which we apply only the LSTM on closing prices.
Secondly, the performance of the model improves including more time-series. Thus, in order to concretely evaluate the model, we compared it with a trivial as much as difficult to beat strategy: $\hat{y}_{t+1} = y_t$. This comparison showed us the misleading nature of the results observed (see Fig. 5.5). Indeed the model is incapable to forecast the behaviour of the target series, in addition it is not able to beat the trivial strategy at every horizon.
From an econometric point of view, the stochastic process underlying the time-series of the closing price is not stationary, i.e. the *mean value* and the *variance* change over time. Such a process is also known as *integrated process* of order p. For example, consider a stochastic process also know as *random walk*. It is defined by:

$$y_t = \theta y_{t-1} + \epsilon_t \tag{5.1}$$

where $\theta = 1$ and $\epsilon_t = IID(0, \sigma_\epsilon^2)$ (*independent and identically distributed* random variable). Therefore, by substituting backward the past observations of y we get:

$$y_t = \underbrace{\sum_{i=1}^{t} \epsilon_i}_{\text{stochastic trend}} + y_0 \tag{5.2}$$

One of the main implication is that $\mathbb{E}[y_t] = y_0$. This is the reason why, in our case, the trivial strategy is so difficult to beat.
Moreover let us consider the first difference of the *random walk*:

$$\Delta y_t \equiv y_t - y_{t-1} = \epsilon_t \tag{5.3}$$

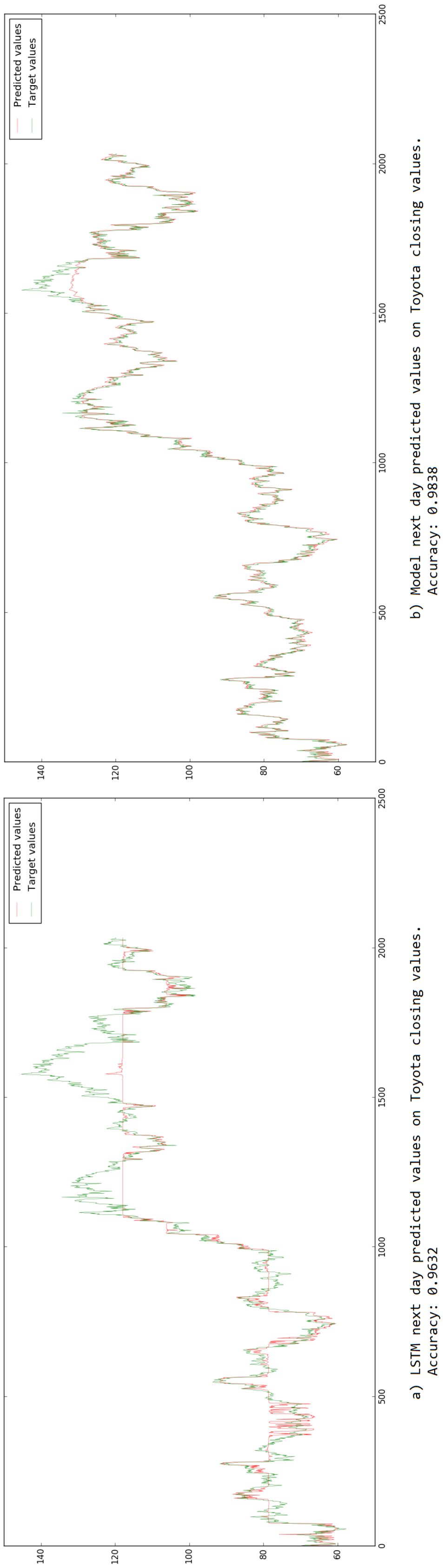
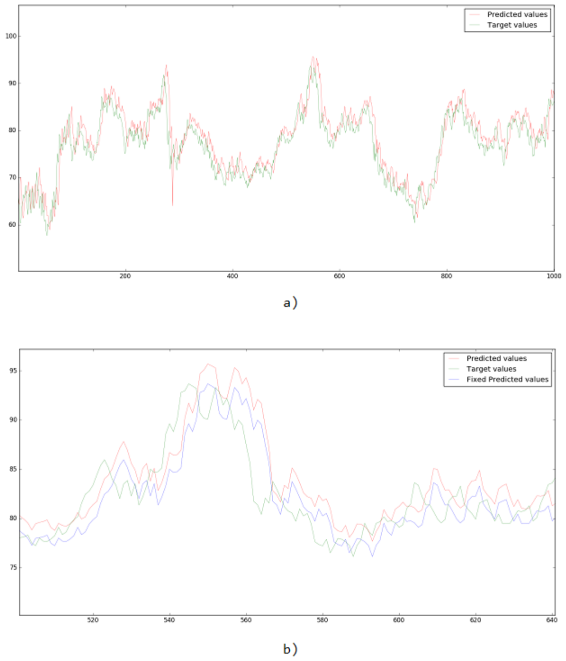


Figure 5.4: The figure shows the performances on the test set of both the LSTM (a) and the proposed model (b). As it is possible to see, we get an important improvement in terms of accuracy as well.



	Accuracy predicted closing values t+1	Accuracy predicted closing values t+2	Accuracy predicted closing values t+3	Accuracy predicted closing values t+4	Accuracy predicted closing values t+5
Using only closing values time series	0.9837	0.9772	0.9757	0.9725	0.9643
Using closing values and exchanged volumes time series	0.9862	0.9810	0.9785	**0.9755**	**0.9736**
Using closing values, exchanged volumes and first differences time series	**0.9868**	**0.9822**	**0.9787**	**0.9755**	0.9725
Strategy $\hat{y}_{t+i} = y_t$	0.9887	0.9843	0.9807	0.9780	0.9756

Figure 5.5: The figure shows the performance on the test set of model in predicting Toyota closing value at $t+5$. As it is possible to see in part a), the model is incapable to forecast the behaviour of the target series. In particular, part b) shows that the values predicted by the model have the same delay of the ones given by the trivial predictor.

Since this process is stationary by definition, the *random walk* is said *integrated process of order 1*:

$$y_t \sim I(1) \tag{5.4}$$

Consequently, the first difference process is defined as an *integrated process of order 0*.

5.3.2 Forecasting differences and returns

By seeking for stationary processes, we analysed the lagged differences and returns, defined by:

- $\Delta^d y_t \equiv y_t - y_{t-d}$ with $d \in \{1, 2, 3, 4, 5, 20\}$;
- $\mathbf{R}^d = \dfrac{y_t}{y_{t-d}}$ with $d \in \{1, 2, 3, 4, 5, 20\}$.

We applied the model to these time-series and we observed improvements w.r.t. the previous case. In particular, using four time-series as input of the deep autoencoder (closing prices, volume exchanged, lagged differences, lagged returns) we were able to beat the trivial strategy as reported in Tab. 5.1 and Tab. 5.2 .
However, by looking at the predicted values on the test set (see Fig. 5.6), we noticed

Predicting differences	MSE 1st	MSE 2nd	MSE 3rd	MSE 4th	MSE 5th	MSE 20th
Model	**2.0132**	**3.9330**	**5.8348**	**7.5786**	**9.1958**	**31.4497**
Null $\Delta = 0$	2.0137	3.9344	5.8359	7.5843	9.2045	31.6058
Gain/Loss	**5e-4**	**1.4e-3**	**1.1e-3**	**5.7e-3**	**8.7e-3**	**1.561e-1**

Table 5.1: Performances of the model in forecasting differences compared to the trivial strategy. In particular, we are able to beat the trivial strategy at each horizon.

Predicting returns	MSE one day	MSE two days	MSE three days	MSE four days	MSE a week	MSE four weeks
Model	2.345e-4	**4.573e-4**	**6.790e-4**	**8.710e-4**	**1.047e-3**	**3.951e-3**
Null $R = 1$	**2.338e-4**	4.578e-4	6.832e-4	8.788e-4	1.057e-3	3.984e-3
Gain/Loss	7e-7	**5e-7**	**4.2e-6**	**7.8e-6**	**1.0e-5**	**3.3e-5**

Table 5.2: Performances of the model in forecasting returns compared to the trivial strategy. In particular, we are able to beat the trivial strategy in each case except $t + 1$.

a tendency to average over the target values. Therefore, we looked at the training of the model to comprehend the reason of this phenomenon.

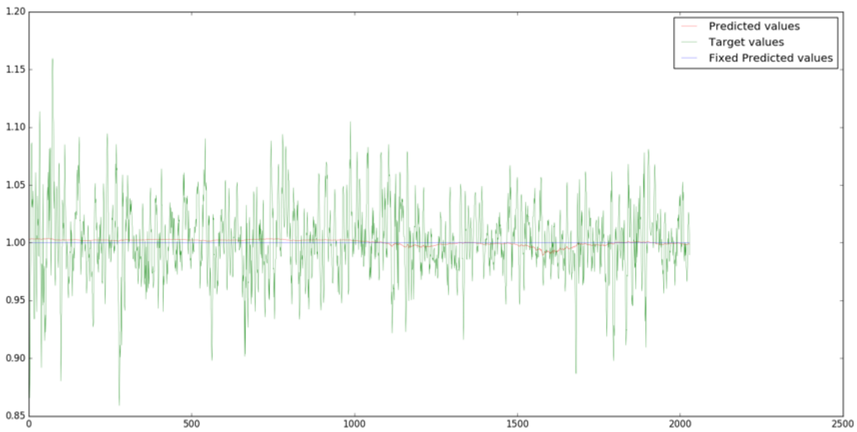


Figure 5.6: Predicting returns at $t + 5$: test set.

As illustrated in Fig. 5.7 (part a), after an initial phase where the parameters are modified to reach the level of the target values, once again the predictions tend to average over the target values.
We attribute the reason of this phenomenon to three main factors:

- sharing parameters, characteristics of the RNN;
- delay of the forecast values w.r.t. the target ones;
- mean reversion of the target series.

More precisely let us consider Fig. 5.7 (part b). The first factor implies that the parameters are shared by all the presented examples, i.e. the optimizer has to change them to reduce the distance between the outputs of the model and the targets. On the other hand, the second and third reason imply the tendency to average over the target values.

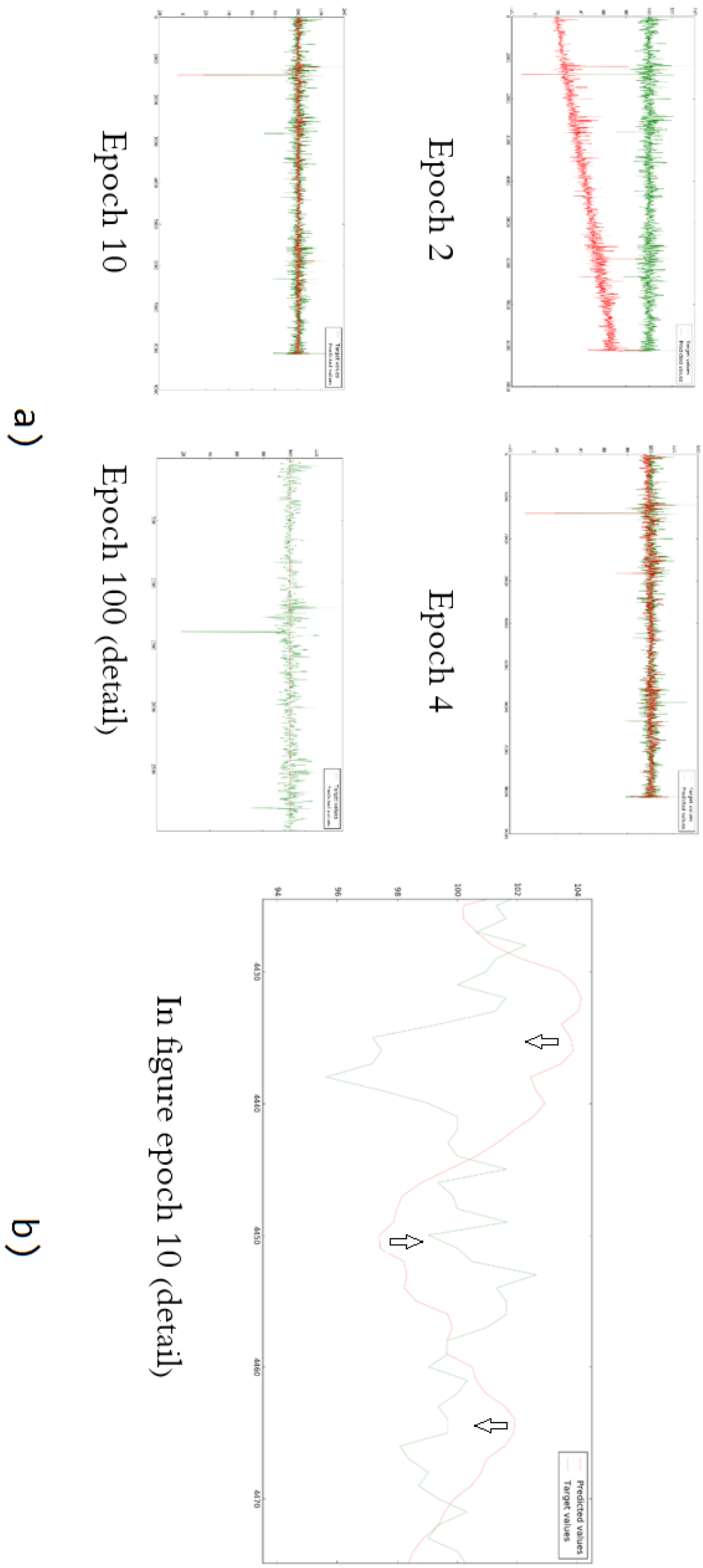


Figure 5.7: Predicting returns at $t + 5$: training.

	t+1	t+2	t+3	t+4	t+5	t+20
Accuracy	0.5170	0.5339	0.5330	0.5401	0.5498	0.5527

Table 5.3: Toyota (NYSE). Dataset: Toyota closing values, volumes exchanged, lagged differences.

	t+1	t+2	t+3	t+4	t+5	t+20
Accuracy	0.5076	0.5290	0.5281	0.5362	0.5409	0.5199

Table 5.4: Toyota (NYSE). Dataset: Toyota closing values, volumes exchanged.

	t+1	t+2	t+3	t+4	t+5	t+20
Accuracy						
Model	0.5042	0.5231	0.5340	0.5361	0.5419	0.5286
LSTM	0.4943	0.5187	5108	0.5067	0.5143	0.5199

Table 5.5: Toyota (NYSE). Dataset: Toyota closing values.

5.3.3 Qualitative analysis

In order to avoid the previous mentioned problems we adapted the proposed model to face the task from a qualitative point of view. The research material related to *Recurrent Neural Network* (RNN) applied to MNIST database, presented in section 3.5, was very useful in order to fulfil this approach. Indeed once again, the extraordinary versatility of deep learning algorithms extremely eased our life in accomplishing this analysis. Substantially the code is the same of section 3.5, this is the reason why we omitted it in the following description; nevertheless we provided the guidelines to understand and reproduce it.
In particular, it was attached to each closing price a label: a **one-hot vector** of size 2 (number of classes) which specifies if the closing value at the time t has increased or decreased w.r.t. the lagged realization $t - i$. In the first case the label will be $[0, 1]$, in the second one $[1, 0]$.
In this way the model forecasts the class whom closing price at time $t + i$ belongs to.
Firstly, we tested this approach on Toyota. The results are shown in Tab.s 5.3 , 5.4 , 5.5 . First of all, as Tab. 5.5 shows, by using the deep autoencoder (and thus the model) as a form of preprocessing we beat the LSTM at every horizon. Moreover reading the three tables all together from the top to the bottom we noticed once again that the three time-series approach leads to the highest performances.
Last but not least the accuracy increases with the horizon, reaching the best value in the case $t + 20$. This is due to the diminishing of randomness and consequently to the emerging patterns.

5.4 Generalizing to NASDAQ stocks

Encouraged by these results we looked at others stocks belonging to different market sectors to check whether it was possible to generalize this qualitative approach. More precisely we paid our attention to 5 stocks of *NASDAQ-100* index: **Apple Inc.**, **Microsoft**, **Texas Instruments Incorporated**, **Intel Corporation**, **Adobe Systems Incorporated**. Other stocks belonging to the same index were discarded because of their small size.
NASDAQ-100 index is made up of 107 equity securities issued by 100 of the largest non-financial companies listed on *NASDAQ*. Moreover, the *Nasdaq Stock Market* is an American stock exchange. It is the second largest exchange in the world by market capitalization, after *New York Stock Exchange* (NYSE). NASDAQ was founded in 1971 and it stays for the acronym of "*National Association of Securities Dealer Automated Quotations*".
Furthermore, we refined our model defining:

- an **indecision zone** whose instances are not discriminated. This area is composed by those closing prices that differ from the closing value at time t of a variation δ, such that $|\delta| < 0.1\%$;
- three **intervals** representing *low*, *medium*, *high* variations.

To realize this refinement, we attached an additional label to the closing price. More precisely, it is a one-hot vector similarly to the label necessary for the training phase, but of size 4. Indeed this vector defines four classes representing the absolute value of the variation, ordered in the following way:

- *low variation*;
- *medium variation*;
- *high variation*;
- *variation belonging to the indecision zone.*

In Fig. 6.1 it is shown an example. The results are shown in the tables below.
Firstly we noticed that the accuracy of the model improves especially for short-term predictions if we consider the indecision zone.
In addition, similarly to Toyota stock, the performance increases with the horizon, reaching the highest values in the case $t + 20$. However, even if the analysis in intervals gives us more information about how the model behaves, it does not seem possible to extract a general rule: the accuracy of the model changes case by case on each interval.

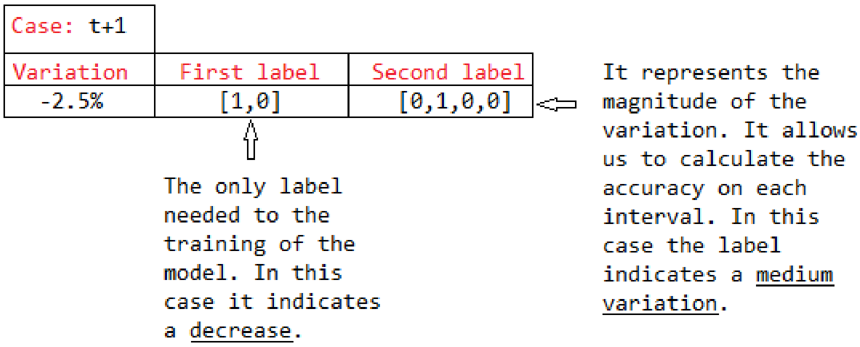


Figure 5.8: Instance of how the two labels are calculated. It is well worth to remark as only the first one is needed to the training of the model. The second allows us to count how many examples in each interval are correctly discriminated. Please notice that if it is $[0, 0, 0, 1]$, the result of the model based on the corresponding example is not included in the calculus of the accuracy.

	t+1	**t+2**	**t+3**	**t+4**	**t+5**	**t+20**
			Intervals			
Low	[0.001,0.01]	[0.001,0.01]	[0.001,0.015]	[0.001,0.02]	[0.001,0.025]	[0.001,0.035]
Medium	(0.01, 0.05]	(0.01, 0.05]	(0.015, 0.055]	(0.02, 0.06]	(0.025, 0.07]	(0.035, 0.10]
High	> 0.05	> 0.05	> 0.055	> 0.06	> 0.07	> 0.10
			Number of examples			
Low	883	633	722	832	915	641
Medium	796	1040	916	781	722	790
High	28	76	115	153	140	345
			Accuracy on each interval			
Low	0.4904	0.5245	0.5111	0.5325	0.5333	0.4774
Medium	0.5653	0.5730	0.5852	0.5839	0.5859	0.6203
High	0.5714	0.5000	0.5043	0.6013	0.5786	0.6725
Accuracy	**0.5267**	**0.5523**	**0.5493**	**0.5612**	**0.5582**	**0.5788**

Table 5.6: Apple Inc. (NASDAQ). Dataset size $\sim$ 10000.

Finally, we tried to forecast the increase or decrease of **Apple Inc.** closing price by using as dataset:

- Apple closing values (technological sector);
- Toyota closing values (automotive sector);
- Helmerich & Payne closing values (energetic sector).

In this way, we tested whether the model captures significant and not intuitive information from the dataset. The results, displayed in Tab 5.11 , show lower accuracies w.r.t. Tab. 5.6 . This means that **volumes exchanged** and **lagged differences** provide more useful information to the model.
However, they are very interesting since, even using three time-series belonging to different market sectors, the model is able to reach a good accuracy in the case $t+20$.

	t+1	t+2	t+3	t+4	t+5	t+20
Intervals						
Low	[0.001,0.01]	[0.001,0.01]	[0.001,0.015]	[0.001,0.02]	[0.001,0.025]	[0.001,0.035]
Medium	(0.01, 0.05]	(0.01, 0.05]	(0.015, 0.055]	(0.02, 0.06]	(0.025, 0.07]	(0.035, 0.10]
High	> 0.05	> 0.05	> 0.055	> 0.06	> 0.07	> 0.10
Number of examples						
Low	853	647	757	848	973	714
Medium	592	808	698	591	483	664
High	13	42	51	65	60	138
Accuracy on each interval						
Low	0.5181	0.5085	0.5337	0.5448	0.5519	0.5532
Medium	0.5000	0.5433	0.5702	0.5753	0.6128	0.5648
High	0.5385	0.5952	0.5686	0.5385	0.4333	0.8043
Accuracy	**0.5110**	**0.5297**	**0.5518**	**0.5565**	**0.5666**	**0.5811**

Table 5.7: Microsoft (NASDAQ). Dataset size $\sim$ 7500.

	t+1	t+2	t+3	t+4	t+5	t+20
Intervals						
Low	[0.001,0.01]	[0.001,0.01]	[0.001,0.015]	[0.001,0.02]	[0.001,0.025]	[0.001,0.035]
Medium	(0.01, 0.05]	(0.01, 0.05]	(0.015, 0.055]	(0.02, 0.06]	(0.025, 0.07]	(0.035, 0.10]
High	> 0.05	> 0.05	> 0.055	> 0.06	> 0.07	> 0.10
Number of examples						
Low	1035	833	964	1119	1228	907
Medium	1026	1195	1069	908	825	961
High	39	123	137	153	134	330
Accuracy on each interval						
Low	0.5324	0.5354	0.5332	0.5648	0.5521	0.5480
Medium	0.5273	0.5389	0.5594	0.5485	0.5588	0.6275
High	0.4615	0.5122	0.5182	0.5163	0.5149	0.5879
Accuracy	**0.5281**	**0.5360**	**0.5452**	**0.5546**	**0.5520**	**0.5887**

Table 5.8: Texas Instruments Incorporated (NASDAQ). Dataset size $\sim$ 11000.

	t+1	t+2	t+3	t+4	t+5	t+20
Intervals						
Low	[0.001,0.01]	[0.001,0.01]	[0.001,0.015]	[0.001,0.02]	[0.001,0.025]	[0.001,0.035]
Medium	(0.01, 0.05]	(0.01, 0.05]	(0.015, 0.055]	(0.02, 0.06]	(0.025, 0.07]	(0.035, 0.10]
High	> 0.05	> 0.05	> 0.055	> 0.06	> 0.07	> 0.10
Number of examples						
Low	936	733	883	1071	1141	694
Medium	788	1000	844	672	609	910
High	13	45	65	48	49	208
Accuracy on each interval						
Low	0.5214	0.5443	0.5413	0.5481	0.5311	0.5562
Medium	0.5190	0.5300	0.5521	0.5446	0.5632	0.5758
High	0.5385	0.4889	0.4615	0.4375	0.5102	0.5865
Accuracy	**0.5204**	**0.5349**	**0.5435**	**0.5438**	**0.5414**	**0.5695**

Table 5.9: Intel Corporation (NASDAQ). Dataset size $\sim$ 9000.

	t+1	t+2	t+3	t+4	t+5	t+20
Intervals						
Low	[0.001,0.01]	[0.001,0.01]	[0.001,0.015]	[0.001,0.02]	[0.001,0.025]	[0.001,0.035]
Medium	(0.01, 0.05]	(0.01, 0.05]	(0.015, 0.055]	(0.02, 0.06]	(0.025, 0.07]	(0.035, 0.10]
High	> 0.05	> 0.05	> 0.055	> 0.06	> 0.07	> 0.10
Number of examples						
Low	748	611	706	769	846	609
Medium	673	796	695	624	569	708
High	23	64	82	95	81	178
Accuracy on each interval						
Low	0.5134	0.5303	0.5340	0.5553	0.5449	0.4910
Medium	0.5468	0.5665	0.5468	0.6074	0.6239	0.6299
High	0.5652	0.5781	0.5366	0.5579	0.5802	0.7303
Accuracy	**0.5298**	**0.5520**	**0.5401**	**0.5773**	**0.5769**	**0.5853**

Table 5.10: Adobe Systems Incorporated (NASDAQ). Dataset size $\sim$ 7500.

	t+1	t+2	t+3	t+4	t+5	t+20
Intervals						
Low	[0.001,0.01]	[0.001,0.01]	[0.001,0.015]	[0.001,0.02]	[0.001,0.025]	[0.001,0.035]
Medium	(0.01, 0.05]	(0.01, 0.05]	(0.015, 0.055]	(0.02, 0.06]	(0.025, 0.07]	(0.035, 0.10]
High	> 0.05	> 0.05	> 0.055	> 0.06	> 0.07	> 0.10
Number of examples						
Low	879	1125	718	823	906	641
Medium	799	548	919	785	725	789
High	27	74	114	155	144	343
Accuracy on each interval						
Low	0.4983	0.5072	0.5056	0.5103	0.4978	0.4851
Medium	0.5282	0.5546	0.5604	0.5669	0.5724	0.6046
High	0.5185	0.4730	0.4649	0.5161	0.5972	0.5802
Accuracy	**0.5126**	**0.5341**	**0.5317**	**0.5360**	**0.5363**	**0.5567**

Table 5.11: Apple Inc. (NASDAQ). Dataset size $\sim$ 9000.

5.5 How should we use the model for trading?

In this section we provide the answer to an important question that is pretty spontaneous if someone would use the proposed model for trading. Indeed the first thing one might ask is whether he/she should act dynamically training the model every day or not. In other words, we wanted to understand if the performance of the model is higher at the beginning of the test set and then it gradually decreases or if it does not degrade over time.
So far we used a test set consisting of several years, thus we split it in **quintiles** and we calculated the accuracies on each of them. A quintile is a statistical quantity representing the 20% of a dataset. More generally it is a type of **quantile**, which is defined as equal-sized segments of a population. Economists often use quintiles to analyse very large datasets.
What we observed is shown in Tab. 5.12 . As it is possible to see, we noticed that the accuracy of the model is prone to degrade over time at each horizon. This is more evident in the cases $t + 5$ and $t + 20$, in which the randomness diminishes.
However it is well worth to highlight another important aspect, emerged when we

	1st quintile	2nd quintile	3rd quintile	4th quintile	5th quintile
t+1	0.5660	0.5308	0.4927	0.5249	0.5191
t+2	0.5788	0.5960	0.4814	0.5817	0.5244
t+3	0.5829	0.5914	0.4714	0.5686	0.5314
t+4	0.5950	0.6006	0.5297	0.5524	0.5269
t+5	0.6056	0.6141	0.4761	0.5521	0.5380
t+20	0.6873	0.6592	0.4113	0.5718	0.5606

Table 5.12: Quintile analysis on the test set. The stock we considered was Apple Inc. (NASDAQ). Dataset size ~ 10000.

analysed what happens in the third quintile, that displayed the lowest accuracies.
In particular it is composed by the data relative to 2013, a difficult and volatile year for the American colossus. During that period Apple Inc. was in crisis: the stock price moved from over 600 dollars per share of the 2012 (a memorable year for the American company) to under 400 dollars per share. This explains the low accuracies of the third quintile: indeed the complexity of the examples presented strongly influences the performance of the model.
So it is reasonable to think that acting in a dynamical way could permit to achieve the highest gains using the proposed machine.

Chapter 6

Conclusion

6.1 Summary

In this dissertation we dealt with a possible application of deep learning to financial analysis. In **Chapter 1** we started our work with a brief analysis of deep learning history and a description of its key elements which have made it one of the most interesting field in recent years. Then we introduced TensorFlow in **Chapter 2** focusing on its main strengths.
Therefore in **Chapter 3** we treasured the notions explained in the previous two sections to deal with three examples of deep learning application to MNIST database. In particular we saw how the use of Python OOP allowed to build and test the three models. Moreover we appreciated the importance of dealing with large-sized datasets: indeed, even though we implemented a simple MLP with three hidden layers, we obtained good performances on MNIST database which is composed by more than 65000 instances.
Furthermore, we immediately noticed the power of a RNN with LSTM cells: in fact, in our case this algorithm was able to beat the CNN which currently holds the state of art on MNIST dataset getting an error rate of 0.21%. As highlighted in different section of this thesis, the three re-elaborations mentioned above were useful to clearly illustrate what was done in the following chapters.
In **Chapter 4** we exploited the versatility of deep learning algorithms. Indeed we built the multilayer structure of the deep autoencoder inspired by the MLP code showed in the previous chapter. Moreover we rearranged the RNNs code of section 3.5 in order to forecast financial time-series.
Then in **Chapter 5** became clear how machine learning models can help us in recognizing market signals even though sometimes the development of this techniques may lead to useless results. For instance, we bumped into this situation in subsection 5.3.1 where we applied the proposed model to forecast directly **Toyota Motor Corporation** closing prices using as inputs three time-series: Toyota closing value, volumes exchanged and lagged differences. Indeed, this was the case in which the architecture was unable to beat the obvious strategy, which asserts $\hat{y}_{t+i} = y_t$. At first glance the results appeared to be astonishing (see part a) of Fig. 5.5) , but actually, the comparison of the model with the trivial predictor led us to analyse the econometric nature of the underlying stochastic process, revealing misleading

outcomes. In spite of failure, we noticed the benefit given by the use of the deep autoencoder as a sort of preprocessing on a single time-series, a significant result throughout the course of this research (see Fig. 5.4).
Since we are interested in stationary processes, we analysed the lagged differences and returns in subsection 5.3.2. We saw how difficulties in the training arise even focusing on this type of processes. Indeed, despite we outperformed the accuracies achieved by the obvious strategy (see Tab.s 5.1 and 5.2), the model showed an anomalous behaviour on both training and testing sets. We identified the reasons of such a phenomenon in three elements:

- sharing parameters, characteristics of the RNN;
- delay of the forecast values w.r.t. the target ones;
- mean reversion of the target series.

Nevertheless those experiments deserve additional research to avoid the model falling in a similar configuration during the training (see Fig. 5.7).
Currently we are making our model larger by using *stacked LSTM*, trained one separately from the other. The intuition is that a structure with two RNNs with LSTM cells, in which the second LSTM takes as inputs the outputs of the first one, might eliminate the delay of the forecasts w.r.t. the targets. However, at the moment the modifications are under development and we still have not achieved significant improvements.
Without a doubt, we appreciated the power of the proposed machine in the qualitative analysis of stock closing prices presented in subsection 5.3.3. Here we forecast if at time $t + i$ Toyota closing value is higher or lower w.r.t. the closing price at t. In this case, the *Efficient Market Hypothesis* (EMH) found demonstrations of its validity especially at short horizons, where the randomness of the process is prevalent. This is due to the impossibility to anticipate information revealed by relevant institutions that influence market behaviour. *Sentiment analysis* may represent a solution to this problem in the near future.
However by increasing the horizon the model started to recognize emerging patterns, reaching the best performances in forecasting the increase or decrease of stock closing prices at $t+20$ (see Tab.s 5.3, 5.4 and 5.5). In fact the machine recognized some market regularities probably arise from the wide use of technical analysis among traders.
Moreover we saw that it is possible to generalize this qualitative approach to other stocks belonging to different market sectors as displayed in section 5.4. Indeed we dealt with five securities belonging to NASDAQ-100 index: **Apple Inc.**, **Microsoft**, **Texas Instruments Incorporated**, **Intel Corporation**, **Adobe Systems Incorporated**. Here used again three time series as inputs: closing price, lagged differences and volumes exchanged relative to each stock. Furthermore we refined our model introducing an indecision zone, whose instances are not discriminated, and defining three intervals associated to low, medium and high variation magnitude.
In this way we obtained even more performing results reaching the 58.87% of accuracy in the Texas Instrument Incorporated case (see Tab. 5.8).

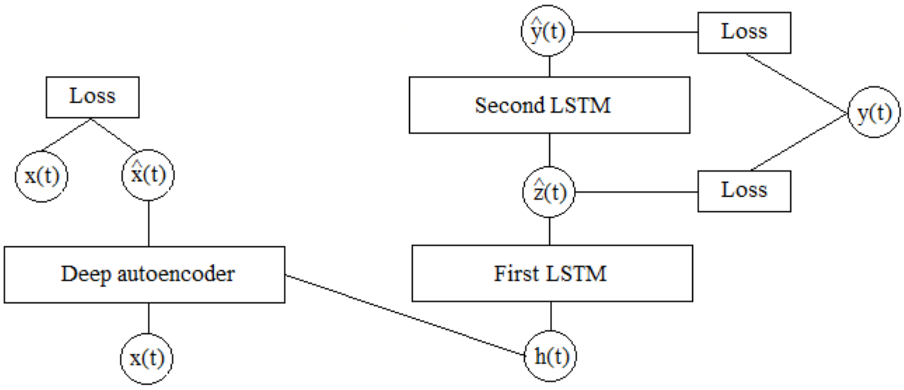


Figure 6.1: Brief illustration of how we are currently modifying the model. We think the addition of the second LSTM might eliminate the delay of the forecasts w.r.t. the targets. Please notice the two LSTMs are separately trained.

Besides we tried to make predictions of Apple Inc. equity based on three different time-series belonging to diverse market sectors: Apple Inc. (technological sector), Toyota (automotive sector) and Helmerich & Payne (energetic sector) closing values. By using these inputs we achieved lower performances (see Tab. 5.11). Thanks to this result we noticed volumes exchanged and lagged differences provided more significant information to the machine.
In the end we divided the test set into quintiles. In this way we concluded that a dynamical strategy could permit to achieve the highest gains using the proposed model for trading.

6.2 Future works

As mentioned above, we are making our model larger implementing stacked LSTM, trained one separately from the other. Currently the modifications are under development and we still have not achieved significant improvements. Therefore, our first purpose is to ultimate this extension and evaluate how model performances improve in this way.
Secondly we would like to continue to investigate the results relative to the forecasting of differences and returns shown in subsection 5.3.2. In particular we are interested in the comparison between the outcomes obtained and several moving averages of different orders to understand to what extent, in terms of orders, lagged and future dependencies matter for the machine. More precisely, our target is to individuate which moving average is most similar to the results returned by the model.
Last but not least we would like to enrich the model making use of sentiment analysis. Sentiment analysis is a very interesting field where both structured and unstructured data are analysed to generate useful insights leading to the improvement of performances. Through text mining of news, blogs, social network and online search results, huge amounts of data are converted into information. Typically this information tells us if the examined data correspond to positive, negative or indifferent

opinions.
When in 2015 the famous Italian novelist and literary critic *Umberto Eco* accused social media of giving freedom of speech to legions of idiots, he ignored that those legions of idiots who create trends opining. For instance, Twitter is considered the ocean of sentiment data and anyone can easily extract material by searching on hash tags. Therefore sentiment analysis has developed as a technology that applies machine learning to make a rapid assessment of the sentiments expressed in new releases.
We think using suitable data relative to the companies considered in this dissertation, such as tweets, official releases from important members companies themselves or important statements from institutions, could significantly improve the predictive power of our model.

I hope this work can inspire future developments, being a starting point for others who deal with deep learning application to finance. Furthermore, my purpose is to carry on the way I have begun investing my competence and creativity to enrich this model.

Bibliography

[1] C. Bagliano (2016). Econometrics II: Univariate Time Series Models. *Lecture notes.*

[2] Y. Bengio, P. Frasconi, and P. Simard (1993). The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1195, San Francisco. IEEE Press.

[3] Y. Bengio, P. Frasconi, and P. Simard (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Tr. Neural Nets.*

[4] J. Duchi, E. Hazan, and Y. Singer (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research.*

[5] F.A. Gers, J. Schimdhuber, F. Cummins (1999). Learning to Forget: Continual Prediction with LSTM. *Neural Computation.*

[6] F. Gers, N. Schraudolph, and J. Schmidhuber. Learning precise timing with LSTM recurrent networks. *Journal of Machine Learning Research*, 2002.

[7] A. Graves (2014). Generating sequences with recurrent neural networks. *arXiv:1308.0850v5[cs.NE]*

[8] G. Hinton (2012). Neural networks for machine learning. *Coursera, video lectures.*

[9] S. Hochreiter (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, T.U. München.

[10] S. Hochreiter, and J. Schmidhuber, (1997). Long short-term memory. *Neural Computation.*

[11] D. P. Kingma, J. L. Ba (2015). Adam: A Method for Stochastic Optimization. *International Conference of Learning Representation (ICLR)*

[12] M. Milano (2004). Guida all'Analisi Tecnica.

[13] H. Sak, A. Senior, F. Beaufays, (2014). Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *INTERSPEECH, 2014.*

[14] J. Schimdhuber (2015). Deep Learning in Neural Networks: an Overview. *Neural Networks.*

[15] N. Srivastava (2013). Improving neural networks with dropout. *PhD thesis*, University of Toronto.

[16] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, R. Fergus (2013). Regularization of Neural Network using DropConnect. *International Conference on Machine Learning.*

[17] D. R. Wilson, and T. R. Martinez (2003). The general inefficiency of batch training for gradient descent learning. Neural Networks, 16 (10), 1429–1451

[18] W. Zaremba, I. Sutskever, O. Vinyals, (2015). Recurrent Neural Network Regularization. *International Conference of Learning Representation (ICLR).*

[19] http://conferences.unicom.co.uk/sentiment-analysis-singapore/

[20] http://www.investopedia.com/terms/q/quintile.asp

[21] http://www.investopedia.com/terms/t/technicalanalysis.asp

[22] http://www.investopedia.com/university/technical/

[23] http://www.focus.it/comportamento/economia/apple-chiude-il-2012-con-numeri-da-capogiro

[24] http://www.huffingtonpost.it/2013/04/23/apple-la-crisi-della-mela-utili-in-calo-perdita-di-valore-in-borsa-pochi-progetti-voci-di-dimissioni-del-ceo-tim-cook_n_3139640.html

[25] http://www.optirisk-systems.com/blog/index.php/sentiment-analysis-applied-to-finance/

[26] https://en.wikipedia.org/wiki/NASDAQ

[27] https://en.wikipedia.org/wiki/NASDAQ-100

[28] https://en.wikipedia.org/wiki/New_York_Stock_Exchange

[29] https://en.wikipedia.org/wiki/Sentiment_analysis

[30] https://en.wikipedia.org/wiki/Technical_analysis

[31] http://stockcharts.com/school/doku.php?id=chart_school:overview:technical_analysis

[32] https://www.tensorflow.org

[33] www.deeplearningbook.org/contents/autoencoders.html

[34] www.deeplearningbook.org/contents/convnets.html

[35] www.deeplearningbook.org/contents/optimization.html

[36] www.deeplearningbook.org/contents/rnn.html