Università
Ca'Foscari
Venezia

Master's Degree
in Economics and Finance

Final Thesis

Artificial Neural Networks and Deep Learning for stress testing a banking system

Supervisor
Ch. Prof. Antonella Basso

Graduand
Krystyna Tsaryk
Matriculation number 856865

Academic Year
2019 / 2020

Table of contents

Introduction 1

CHAPTER 1 5

Artificial Neural Networks 5
 1.1 ANNs characteristics and history 5
 1.2 Application to challenging problems 7
 1.3 Biological and artificial neurons 9
 1.4 Architectures and learning paradigms 12
 1.5 A simple NN model: The Perceptron 13
 1.6 From Perceptron to ADALINE and MLP 16
 1.7 Other popular ANNs 37
 1.8 ANNs compared to statistics and expert systems 38

CHAPTER 2 41

The development of an ANN 41
 2.1 Steps in an ANN development 41
 2.2 Related issues and possible solutions 43

CHAPTER 3 53

Deep learning and deep neural networks 53
 3.1 Deep learning characteristics and history 53
 3.2 Reasons why DNNs are difficult to train 56
 3.3 Convolutional networks 60
 3.4 LSTM networks 71
 3.5 Other popular Deep learning models 79
 3.6 Techniques for improving training 83

CHAPTER 4 89

Banking systems 89
 4.1 Systemic risk and financial stability 89
 4.2 Stress-testing for financial stability 92

CHAPTER 5 101

Deep Learning applications for banking systems 101
 5.1 Financial applications of ANNs and DNNs 101
 5.2 Credit risk evaluation and credit scoring 102
 5.3 Bank insolvency and bankruptcy prediction 105
 5.4 Financial crisis prediction and early warning systems 108
 5.5 Dynamic balance sheet stress-testing 110

CHAPTER 6 113

Application to dynamic balance sheet stress-testing 113
    6.1 Data collection and pre-processing 113
    6.2 Model development and hyperparameters tuning 119

Conclusions and closing remarks 137

Bibliography 139

Appendix A 145

A list of the codes for the case study 145
    A.1 Outliers' detection and scaling choice 145
    A.2 Initial trial 147
    A.3 Learning curve for the training set size 149
    A.4 Learning curve for the batch size 151
    A.5 Learning curve for the learning rate 153
    A.6 Learning curve for the number of hidden neurons 156
    A.7 Learning curve for the weight decay 158
    A.8 Choice of the momentum 160
    A.9 Choice of the dropout probability 162
    A.10 Choice of the dropout probability with L2 regularization 165

Introduction

The aim of this thesis is to review how artificial neural networks and deep learning techniques can be applied for solving different types of financial problems that can be encountered by banks and regulatory authorities. In particular, a broad review about neural networks and an introduction to stress-testing concepts is provided, together with a final case study analysis to put into practice the notions learnt.

Stress tests are widely used by regulatory authorities to assess the financial stability of the single institutions and the whole financial system, as well as by banks for internal risk-management and self-assessment purposes. They are important risk-management tools carrying essential information on the health of the system and on the vulnerabilities to extreme but plausible events. After the global financial crisis of 2007 and 2008, stress tests became a crucial component of the supervisory and financial stability toolbox of central banks. Both the European Banking Authority and the European Central Bank conduct periodically these tests to assess the individuals' banks resilience and the system wide impact of the adverse shocks.

The financial health of banks during an adverse shock depends on several factors, such as capital adequacy, asset quality, management capability, earnings, liquidity and sensitivity. After the lessons learnt from the financial crisis, the Basel Committee on Banking Supervision improved its regulatory standards and guidelines contained in the Basel frameworks. The supervisory authorities and regulators follow these standards closely as a guidance for an adequate capital and liquidity allocation and monitor the banks so that they can withstand possible systemic shocks.

Several real-world financial problems have a non-linear behavior, which is difficult to capture with classical statistical tools. For this reason, a growing interest has risen in machine learning techniques and especially in artificial neural networks.

ANNs are computational modelling tools inspired by biological nervous systems. They are composed by a set of processing units that are highly interconnected and able to perform complex parallel computations, while both acquiring and keeping new knowledge. From the simplest architecture of the Perceptron made of only one neuron, more complex networks were developed in the recent years. These are called deep neural networks and are characterised by a multi-layered architecture comprising two or more hidden layers and several hidden neurons in each layer. Multilayer ANNs are powerful tools, able to discover complex and nonlinear functional mappings. They are built of multiple processing layers that are able to learn data representations by using multiple levels of abstraction that describe the degree of complexity. All these characteristics make ANNs clearly appealing when dealing with financial problems. This work reviews the possible financial applications, with a particular focus on stress-testing for which there is still limited empirical researche and future investigations can be prompted. ANNs could offer significant advantages if they were used in conjunction with the classic stress-testing tools to support the supervisory activity. They could increase the predictive accuracy of the estimates by recognizing complex mappings in the dataset that classic statistical tools may not be able to grasp.

The present thesis is structured as follows.

Chapter 1 provides a broad review about artificial neural networks, with a particular focus toward their key characteristics and their similarities with biological neural systems. Some popular and simple models, such as the Perceptron, the ADALINE and the Multilayer Perceptron, are discussed in order to have a clear understanding of their functioning. The last section of the chapter is dedicated to a comparison of neural networks with statistics and expert systems, together with some suggestions on the choice of the adequate tool.

Chapter 2 deals with presenting and discussing the methodology and the required steps that have to be followed in order to develop a new ANN project. Many issues relating to the dataset, the architecture and the training of the model have to be accounted for, when developing an ANN. A summary of the issues and possible solutions is presented in the second section of the chapter.

Chapter 3 is dedicated to analysing deep learning and the potential of deep neural networks. The main characteristics that differentiate deep architectures from shallow ones are also discussed. The following sections reason on why deep neural networks are hard to train and how researchers can address this issue. Two popular architectures, namely convolutional neural networks and long short-term memory networks are presented in the last sections.

Chapter 4 introduces two important subjects related to banking systems. The first part of the chapter discusses the issue of systemic risk and its impact on financial stability. The second part illustrates stress-testing as a tool for assessing the financial stability of a banking system.

Chapter 5 discusses how the use of artificial neural networks can improve several real-world financial applications that have a non-linear behaviour. Some of the most important areas of application of ANNs and deep learning to banking systems are presented, namely credit scoring, bankruptcy prediction, financial crisis prediction and stress-testing.

Chapter 6 contains a final case study analysis in which the application of a neural network to dynamic balance sheet stress-testing is performed by using real US data. At the same time, the choice of a suitable architecture and hyperparameters is examined with the purpose of enhancing the model's generalization and predictive capabilities.

CHAPTER 1
Artificial Neural Networks

This thesis aims at investigating the possible applications for the banking sector of artificial neural networks, which comprise several empirical modelling methods that show an excellent performance with real-world data and can dramatically outperform classical statistical tools. These methods are more challenging to use and understand than statistical tools. For this reason, the first chapter is dedicated to a rich overview of ANNs that will ensure a sound knowledge, essential to the comprehension of more complex deep networks that will be presented and applied to financial problems in the following chapters.

The present chapter will present a broad review about artificial neural networks, with a particular focus toward their key characteristics and their similarities with biological neural systems. Some popular and simple models, such as the Perceptron, the ADALINE and the Multilayer Perceptron, will be discussed in order to have a clear understanding of their functioning. A practical application for solving simple classification problems using ADALINE and MLP will be also shown together with their related codes that can be used in R. The last section of the chapter will be dedicated to a comparison of neural networks with statistics and expert systems, together with some suggestions on the choice of the adequate tool.

1.1 ANNs characteristics and history

Artificial neural networks (ANNs) are computational modelling tools inspired by biological nervous systems and are able to solve many complex problems of the real world. An ANN is a structure composed by a set of processing units (called artificial

neurons, nodes or units) that are highly interconnected and able to perform complex parallel computations.

The ability to process information, while both acquiring and keeping new knowledge, is the basis of several characteristics concerning nonlinearity, high parallelism, robustness, fault and noise tolerance, learning and adaptation, handling imprecise and fuzzy information and their capability to generalize. Better fitting of the data can be achieved with a non-linear model, an increased processing speed is reached with high parallelism, a more accurate prediction even in case of uncertain or erroneous measures can be achieved with insensitivity to noise, the internal structure can be easily modified when the external environment changes when the system is able to learn and the model can be applied to unknown information due to its generalization capability.

The learning ability is the most distinctive characteristic of a neural network and it can be defined as an iterative adjustment of its internal architecture as it receives different external stimuli. This adjustment enables the system to acquire knowledge by experience and thus to able to handle noisy information with good accuracy and to estimate solutions so far unknown (Basheer and Hajmeer, 2000).

The first step toward ANNs was made in 1943 by McCulloch and Pitts. They constructed the first network with electrical circuits, based on a mathematical model inspired by biological neurons. A few years later, in 1949 Hebb proposed the first method for training an artificial neural network. This rule, named Hebb's learning rule, was based on the observation of biological neuron's synapse

At the same time, during the 1950s it became possible to model theories concerning human thought thanks to computer advancements. Between 1957 and 1958 the neurophysiologist Rosenblatt at Cornell University was inspired by the operation of the eye of a fly and developed the first successful neurocomputer called Mark I Perceptron, able to perform simple character recognition.

In 1959 Widrow and Hoff at Stanford University developed two models called ADALINE and MADALINE, which were based on adaptive linear elements. The learning of those models was based on the Delta rule, rather than the Hebb's rule used with the Perceptron. MADALINE was the first ANN to be addressed to a real-world problem and it was utilized as a filter to eliminate echoes on phone lines.

The earlier success caused an excessive hype about neurocomputing, particularly considering the limitations in the computers available at that time. This hype suffered a major slowdown with the publication of the book *Perceptrons* by Minsky and Papert in 1969 in which they overexaggerated the limitations of the Perceptron and their inability to solve nonlinear classification problems. Following this publication, researches on neural networks were greatly reduced, due to the lack of funding.

Few important studies were made during the 1980s and caused a renewed interest in the field. In 1980 Grossberg developed the Adaptive Resonance Theory network, in 1983 Kohonen formulated the self-organized maps and in 1984 Hopfield introduced recurrent networks based on energy functions, called Hoppfield networks. In 1986 Rumelhart, Hinton and Williams brought to light the backpropagation algorithm, that performed weights adjustment for multilayer networks. The comeback of ANNs was clear when in 1987 the first annual IEEE International ANNs Conference was created. Shortly after the International NN Society was formed and their started publishing their own journal.

In recent years many new researches have been made and brought to some important theoretical advancements and practical applications to the real world.

To learn further about the historical digression, see Anderson and McNeill (1992), Basheer and Hajmeer (2000) and Da Silva et al. (2017).

1.2 Application to challenging problems

Artificial neural network models are empirical, however, due to their learning ability, they provide accurate and robust solutions for both precisely and imprecisely formulated problems. For this reason, they have been employed in several problems. The potential application areas range from pattern classification, clustering, modelling, forecasting, optimization, association and control.

The application categories in detail are:

- Pattern classification or recognition is utilized for associating unknown input patterns to one of the several previously defined classes, based on properties

common to the related class. Typical examples are image, writing and speech recognition;

- Data clustering goal is to detect similarities and dissimilarities between several input patterns based on their intercorrelations and to group them accordingly. Example applications include data mining and automatic class identification;
- Modelling (function approximation) aim is to approximate the functional relationship between inputs and outputs starting from a meaningful set of known values. Usually it is used to map processes for problems where it is not feasible to use traditional methods or when a theoretical model is not available at all;
- Forecasting involves the prediction of future values for a process from several previous historical observations. Typical applications can be found for time series prediction, stock prices movements and forecast about weather conditions;
- Optimization aims to maximize or minimize an objective function obeying to a set of constraints. ANNs are especially efficient instruments for solving complex nonlinear optimization problems and are frequently employed in constrained optimization problems and dynamic programming;
- Association is concerned with training a model which is able to classify, recover and correct corrupted or partially missing data, after it has learned to classify data free of noise. Typical applications relate to image processing, transmission of signals and identification of characters;
- Control relates to devising a network that works in conjunction with an adaptive control system in order to produce control inputs that will meet a set of specific requirements. Examples pertain to airplanes, robotics and satellites functioning[1].

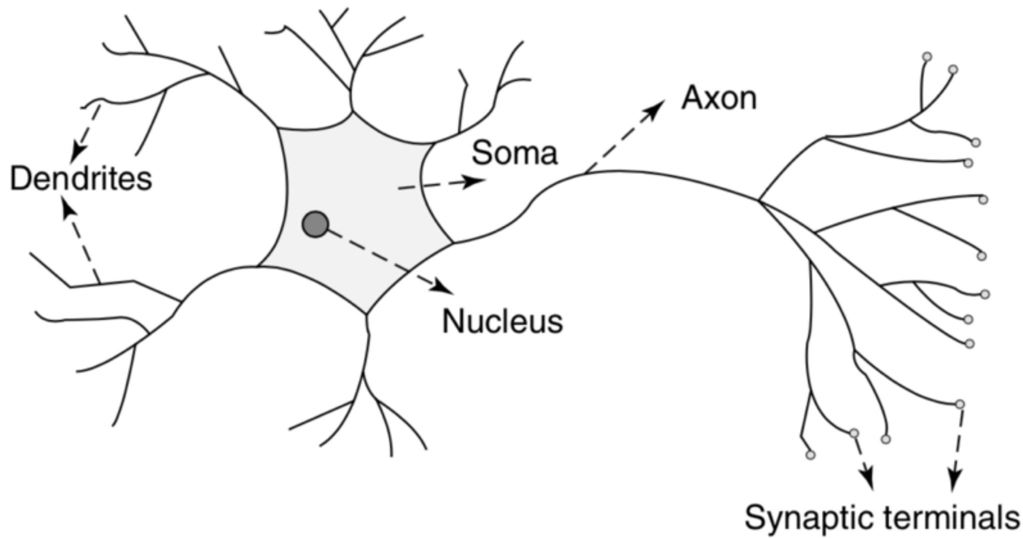
[1] For further details about ANN applications see Basheer and Hajmeer (2000) and Da Silva et al. (2017)

1.3 Biological and artificial neurons

The human nervous system consists of more than 10 billion interconnected neurons functioning together and manged by the brain. The human brain is responsible for executing cognitive (acquiring knowledge), emotional (language ability and identification of faces) and control (movement of the body and functioning of the organs) tasks. It is able to handle computationally demanding activities by using a highly parallel structure, which allows the brain to process imprecise and incomplete pieces of information.

The fundamental cell of the nervous system is the neuron and its role is to receive, process and conduct information in the form of electrical impulses or stimuli by using biochemical reactions.

Figure 1.1. Schematic of biological neuron



Source: Abraham (2015)

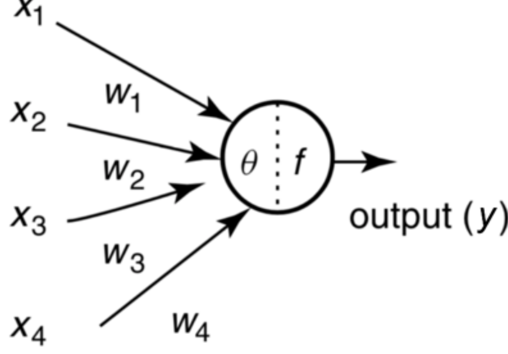
The biological neuron can be divided into three main functional units, as can be seen in Figure 1.1: dendrites, soma (or cell body) and axon. The dendrites are responsible for receiving and passing over impulses from other neurons or the external environment to the soma. The soma contains the cell nucleus and is responsible for processing the information and for generating an activation potential. The axon is responsible for guiding the impulses from the soma to the dendrites of neighbouring neurons through synaptic terminals or synapses. Electric

signals are transmitted through a complex chemical process. Specific neurotransmitters are released from the synapses in order to change the electrical potential of the receiving cell and propagation is triggered if the potential threshold is reached (Abraham, 2015).

ANNs were developed as abstractions or generalizations of mathematical models of their biological counterparts in order to mimic the computational properties of the human brain, such as information processing and knowledge acquisition. The idea behind the artificial networks is not the replication of the operations of the biological nervous system, but rather employing its efficiency in solving complex problems.

The basic processing units or computational components of an artificial neural network are the neurons, called also nodes. In the simplified model shown in Figure 1.2, it can be seen the functioning of an artificial neuron: the neuron receives input signals $\{x_1, x_2, \dots, x_n\}$ from the external environment, each with a different intensity based on the connection (or synaptic) weights $\{w_1, w_2, \dots, w_n\}$ representing the synapses. The weighted sum of the input signals is computed by a linear aggregator in order to form the net ξ. Then the net is compared to the threshold (or bias) and the result will be the activation potential u. If the threshold is reached, the neuron will become activated and it will produce an output signal y, applying a specific activation (transfer) function f to the result.

Figure 1.2. Architecture of an artificial neuron



Source: Abraham (2015)

Activation functions are important for limiting and scaling the final output in order to control for its scale. Table 1 presents some commonly used transfer functions that I summarized from Da Silva et al. (2017). In this table $g(\,)$ is the activation function and it is applied to the activation potential u.

Table 1.1. Common activation functions

Activation function	Input/output relation	Graph
Step	$g(u) = \begin{cases} 1, & if\ u \geq 0 \\ 0, & if\ u < 0 \end{cases}$	
Bipolar step	$g(u) = \begin{cases} 1, & if\ u > 0 \\ 0, & if\ u = 0 \\ -1, & if\ u < 0 \end{cases}$	
Symmetric ramp	$g(u) = \begin{cases} a, & if\ u > a \\ u, & if\ -a \leq u \leq a \\ -a, & if\ net < -a \end{cases}$	
Logistic (sigmoid)	$g(u) = \dfrac{1}{1 + e^{-u}}$	
Hyperbolic tangent (sigmoid)	$g(u) = \dfrac{1 - e^{-u}}{1 + e^{-u}}$	
Linear	$g(u) = u$	

Hard limit, bipolar step and symmetric ramp are all partially differentiable activation functions. On the other hand, logistic, hyperbolic tangent and linear functions are fully differentiable on their entire domain of definition.

1.4 Architectures and learning paradigms

ANNs can be classified in many different ways depending on the set of features considered. A first distinction can be done according to the type of problem that the network is devised to handle. Section 1.2 described the possible applications, such as pattern classification, clustering and modelling.

A second distinction is related to the degree of connectivity between the various neurons. They can be fully connected so that every node is connected to the other nodes or they can be partially connected.

Another distinction can be based on the direction in which the information flows, which can be feed-forward or recurrent. In a feed-forward network, signals flow in a single direction from the input to the output and thus no feedback is present. Recurrent networks on the other hand are dynamic systems in which the output at any given time depends on the inputs but also on previous outputs.

The type of learning algorithm is also an important feature to consider. It represents a set of ordinated steps and systemic equations that adjust weights and thresholds of the network and updates the internal structure of the network until the outputs obtained are close enough to the desired values. Some important learning rules such as Hebb's learning rule, Delta rule and Backpropagation learning rule will be discussed in section 1.5 and 1.6 in relation to the Perceptron, ADALINE and Multilayer Perceptron.

In relation to the learning algorithm it must be also specified the learning rule, which is its primary factor. The learning rule defines the specific procedure to follow for adjusting the network weights between each epoch (a successive training cycle). There are four types of rules. The first is the error-correction learning (ECL) rule,

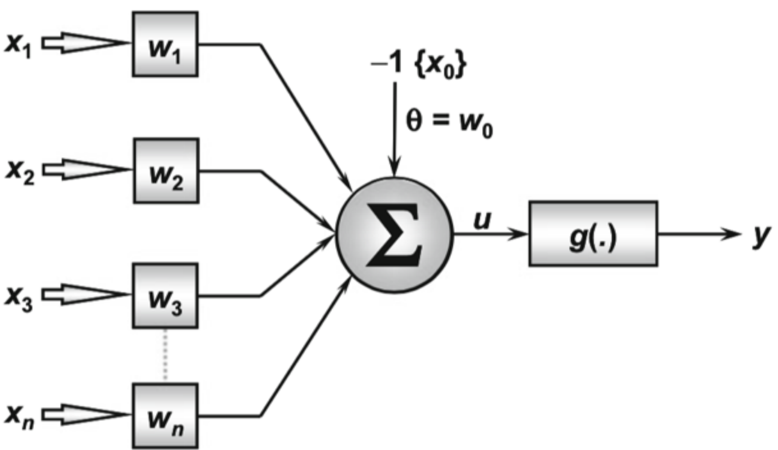
which takes the arithmetic difference in each epoch between the obtained outputs and desired values. The error computed in this way is used to adjust the weights and to gradually reduce the overall error. The second rule is the Boltzman learning (BL) rule, a stochastic rule derived from thermodynamic theory. In this setting, the outputs generated are based on a Boltzman statistical distribution and the difference between the probability distributions of the system is taken rather than the difference between desired and generated outputs. The learning is much slower than with the ECL rule. The third rule is the Hebbian learning (HL) rule, based on neurobiological experiments. In this case learning is performed locally by adjusting weights based on activity of neurons. If two neurons are active at the same moment, their interconnection should be strengthened. The fourth and final rule is the competitive learning (CL) rule, in which all neurons compete among themselves. Only one neuron will be activated in each iteration and its weights will be adjusted.

The last feature to take into account is the degree of learning supervision needed, that can be supervised, unsupervised or reinforcement learning. Supervised learning involves presenting to the ANN both the input vector together with a set of desired responses (or target outputs). The difference between the target and ANN solution is then used to adjust the weights. Unsupervised learning does not require to be fed a priori with the target outputs. The system will develop its own representation of the stimuli, through an examination of the underlying structure and the correlations in the dataset and by organizing the examples into clusters based on their similarity. Reinforcement learning is a form of supervised learning, however the system is not provided with a desired response, but a numerical reward signal. The ANN thus will discover the combination of weights giving the highest reward by a trial and error search (Basheer and Hajmeer 2000).

1.5 A simple NN model: The Perceptron

The Perceptron is the simplest design of an ANN and it is composed by a single artificial neuron. Figure 1.3 presents the Perceptron network, which is composed of n input signals and one final output.

Figure 1.3. Perceptron network



Source: Da Silva et al. (2017)

The neuron receives input stimuli from the external environment and takes the weighted sum of inputs to form the net $\xi = \sum_{i=1}^{n} w_i \cdot x_i$, where x_i is the *i*th input coming from the signal flow $x_1, x_2, \dots, x_n$ and w_i is the weight associated with the *i*th input. The neuron then compares the net to the threshold and verifies if $\xi \geq \theta$. The activation of the neuron will be triggered only if the net exceeds the threshold and in this case the output y will be 1. Otherwise, the output resulting from the activation function will be 0.

From a formal point of view, this inner processing can be described as:

$$\begin{cases} u = \xi - \theta \\ y = g(u) \end{cases} \tag{1.1}$$

where u is the activation potential, θ is the threshold, ξ is the net and $g(\,)$ is the activation function.

Typical transfer functions used for the Perceptron are the step and bipolar step functions. In case of the step function, output y can be written as:

$$y = \begin{cases} 1, & if \;\; u \geq 0 \\ 0, & if \;\; u < 0 \end{cases} \tag{1.2}$$

The separation between input classes, made by the Perceptron, can be seen also from a different point of view. The Perceptron behaves as a pattern classifier and is able to partition linearly separable input classes by splitting the input space with

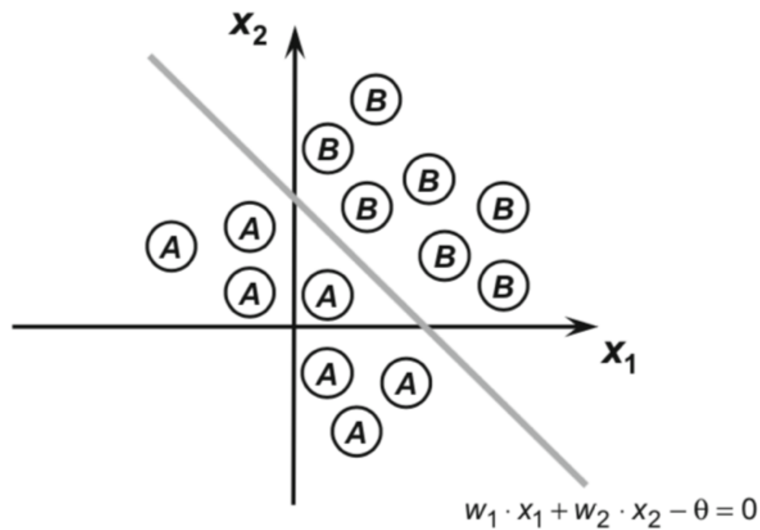
hyperplanes into different regions. The decision boundary that shatters the space is given by the equality:

$$\sum_{i=1}^{n} w_i \cdot x_i - \theta = 0 \quad (1.3)$$

In case of two inputs, the decision boundary is a straight line (*one*-dimensional), for three inputs, it is represented by a plane (*two*-dimensional) and for higher dimensions of order *n*, it will be a hyperplane (*n-1*-dimensional).

The simplest case can be seen in Figure 1.4, where the decision boundary for a neuron with two inputs classes (class A and class B) is illustrated as a straight line given by $w_1 \cdot x_1 + w_2 \cdot x_2 - \theta = 0$. Patterns belonging to class A will be located below the decision boundary and patterns belonging to class B will be above the boundary.

Figure 1.4.Decision boundary with two inputs



Source: Da Silva et al. (2017)

The training process of the perceptron is made through supervised learning. Weights and threshold are adjusted according to Hebb's learning rule. In short, the network is initialized with random weights and threshold and an input vector is selected from the training set. If the output coincides with the desired value, weights and threshold remain unchanged, otherwise if it is different from the desired value, then weights and threshold will be adjusted according to:

$$\boldsymbol{w}_i^{current} = \boldsymbol{w}_i^{previous} + \eta \cdot \left(d^{(k)} - y\right) \cdot \boldsymbol{x}_i^{(k)} \tag{1.4}$$

where $\boldsymbol{w} = [\theta \ \ w_1 \ \ w_2 \dots w_n]^T$ is the vector containing current threshold and weights, $\boldsymbol{x}^{(\boldsymbol{k})} = [-1 \ \ x_1^{(k)} \ \ x_2^{(k)} \dots x_n^{(k)}]^T$ is the kth training sample, $d^{(k)}$ is the desired value for the kth training sample, y is the output produced by the Perceptron and η is the learning rate. The change in the weight vector can be written from the previous equation as:

$$\Delta \boldsymbol{w_i} = \eta \cdot \left(d^{(k)} - y\right) \cdot \boldsymbol{x}^{(\boldsymbol{k})} \tag{1.5}$$

The learning rate defines how rapidly the training process will converge and become stable. Usually it is a number within the range $0 < \eta < 1$. The adjustment process is repeated sequentially until no error is found between the desired and actual output values[2].

1.6 From Perceptron to ADALINE and MLP

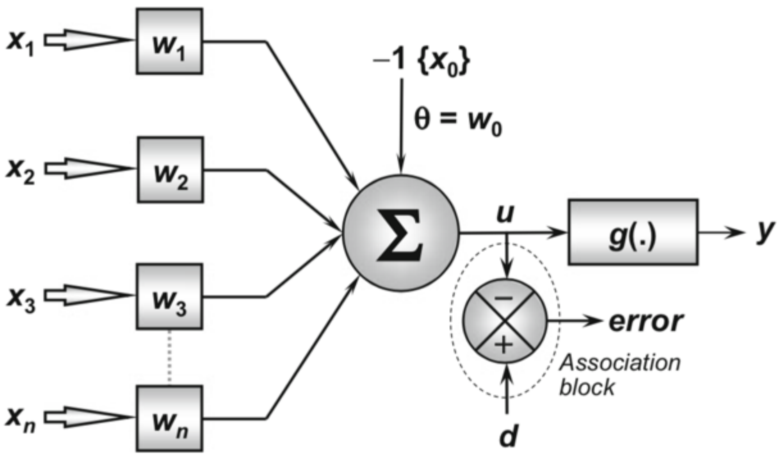
ADALINE network

The ADELINE (adaptive linear element) is similar to the Perceptron in its structure, since it is also composed by one artificial neuron that receives n input signals and produces one output. However, it is characterised by an essential improvement regarding a different learning algorithm, which is the Delta rule.

From Figure 1.5 it can be seen that the operating principle of the ADALINE is similar to the Perceptron, but with the addition of an association block, which is responsible for producing an error signal during the training process. The error signal is simply the difference between the activation potential u and the desired output d.

[2] For further details about the mathematical analysis of the Perceptron see Da Silva et al. (2017) pp. 29-40

Figure 1.5. ADALINE architecture



Source: Da Silva et al. (2017)

The weights and the threshold are adjusted applying the Delta rule, which belongs to the family of Gradient Descent (GD) learning rules. GD algorithms approach the minimum of a function by taking steps in the opposite direction of the gradient of the function. Delta learning uses the minimum of the error for the adjustment process.

To quantify the error and measure the appropriateness of the transfer function, a loss function is considered. In detail, the squared-error loss function is used, because it is a convex function and it is possible to find its minimum. Since the probability distribution of the loss function cannot be directly computed, the empirical risk (or error) is estimated based on the training sample. The objective of the network is to converge to the minimum of the empirical error function so that the squared error is as small as possible for all p training samples available. The squared error can be formulated as:

$$E(\boldsymbol{w}) = \frac{1}{2}\cdot\sum_{k=1}^{p}[d_k - u_k]^2 = \frac{1}{2}\cdot\sum_{k=1}^{p}[d_k - \sum\nolimits_{i=1}^{n}(w_i \cdot x_i - \theta)]^2 \qquad (1.6)$$

To find the minimum, the derivative (gradient) of the squared error $E(\boldsymbol{w})$ with respect to the vector $\boldsymbol{w}$ is taken:

$$\nabla E(\boldsymbol{w}) = \frac{\partial E(w)}{\partial w} \qquad (1.7)$$

The variation $\Delta \boldsymbol{w}$ needed to update the weight vector will have the opposite direction of the gradient, according to:

$$\Delta \boldsymbol{w} = -\eta \cdot \nabla E(\boldsymbol{w}) = \eta \cdot \left(d^{(k)} - y\right) \cdot \boldsymbol{x}_i^{(k)} \quad (1.8)$$

The weights and the threshold will be thus adjusted according to:

$$\boldsymbol{w}_i^{current} = \boldsymbol{w}_i^{previous} + \eta \cdot \left(d^{(k)} - y\right) \cdot \boldsymbol{x}_i^{(k)} \quad (1.9)$$

where $\boldsymbol{w} = [\theta \ w_1 \ w_2 \dots w_n]^T$ is the vector containing current threshold and weights, $\boldsymbol{x}^{(k)} = [-1 \ x_1^{(k)} \ x_2^{(k)} \dots x_n^{(k)}]^T$ is the kth training sample, $d^{(k)}$ is the desired value for the kth training sample, y is the output produced by the ADALINE and η is the learning rate.

Weights are adjusted until the process converges to the optimal configuration of the internal parameters. The convergence criterion employs the mean squared error $\bar{E}(\boldsymbol{w})$ with respect to all training samples and can be computed as:

$$\bar{E}(\boldsymbol{w}) = \frac{1}{p} \cdot \sum_{k=1}^{p} [d_k - u_k]^2 \quad (1.10)$$

The stopping criterion is based on the difference between the current and the previous mean squared error as follows:

$$\left|\bar{E}(\boldsymbol{w}^{current}) - \bar{E}\left(\boldsymbol{w}^{previous}\right)\right| \leq \varepsilon \quad (1.11)$$

When this difference becomes lower than the required precision ε, convergence will be reached.

The ADALINE and the Perceptron will provide different results, since they are based on two different learning rules. The Perceptron decision boundary will be different each time the model is trained, considering that the weight vector strongly depends on its random initial values. The ADALINE, however, will always reach the same optimal configuration since the configuration of the optimal parameters has the minimum error. For all those reasons, ADALINE is a more robust model with respect to eventual noise elements in the sample[3].

[3] For further details about the mathematical analysis of the ADALINE see Da Silva et al. (2017) pp. 41-54

Coding the ADALINE in R

It is possible to write a code in R to implement a simple classification task in which a single input variable x_1 produces an output belonging to the set $\{0,1\}$. The set of R functions in Listing 1.1 contains the activation function, the ADALINE training function returning as a result the optimal weights and threshold, the ADALINE test function classifying new examples and a final function performing training on given training and test sets. The codes presented in this section are based on the algorithms presented in De Mello and Ponti (2018)[4].

The weight and the threshold are randomized at the beginning, then the network adjusts them performing the Gradient Descent method on an example basis with an iterative procedure until the SSE converges to 0. The expected and produced outputs are shown for the training set in addition to the optimal weight and threshold.

Listing 1.1. ADALINE.r code

```
# define the activation function (a step function)
g <- function(net) {
  if (net > 0) {
    return(1)
  }else{
    return(0)
  }
}

# define the function to train the ADALINE
# eta and epsilon assume default values, but can be
# changed to desired values
ADALINE.train <- function(train.table, eta = 0.15,
                             epsilon = 0.1) {

  # define number of input variables (nVars)
  nVars = ncol(train.table)-1
  cat("Randomizing weights and theta in range [-0.6, 0.6]\n
      ")
```

[4] To deepen the knowledge about the codes, see De Mello and Ponti (2018) pp. 28-52

```
# randomize weights and theta
weights = runif(min = -0.6, max = 0.6 , n = nVars)
theta = runif(min = 0.6, max = 0.6 , n = 1)

# initialize the sum of errors that will contain all training errors
# when the error falls below the precision rate, learning stops
sumSquaredError = 2*epsilon

# learning iterations
while (sumSquaredError > epsilon) {

  # initialize the SSE as zero to start counting
  sumSquaredError = 0

  # iterate along all examples contained in train.table
  for (i in 1:nrow(train.table)) {

    # example (x_i)
    x_i = train.table[i, 1:nVars]
    # target output class (y_i)
    y_i = train.table[i, ncol(train.table)]
    # outup produced by ADALINE (hat_y_i)
    # applying activation function with current weights and theta
    hat_y_i = g(x_i %*% weights - theta )
    # compute error
    Error = y_i-hat_y_i
    # compute the partial derivative of the squared error
    # using Gradient Descent method on an example basis
    dE2_dw1 = Error * x_i
    dE2_dtheta = Error * -1
    # find new weights and theta
    weights = weights + eta * dE2_dw1
    theta = theta + eta * dE2_dtheta
    # accumulate SSE to define stop criterion
    sumSquaredError = sumSquaredError + Error ^2
    }
  cat("Sum of squared errors = " , sumSquaredError , "\n")
  }

# return final weights and theta as solution
ret = list ()
ret$weights = weights
```

```
  ret$theta = theta
  return(ret)
}

# define the function to test ADALINE over unseen examples
ADALINE.test <- function(test.table , weights , theta) {

  # print target and produced outputs
  cat("#yi\that_yi\n")
  # define the number in inputs
  nVars = ncol(test.table)-1

  # compute network outputs with weights and theta previously found
  for (i in 1:nrow(test.table)) {

    # example i
    x_i = test.table[i, 1:nVars]
    # target class for example i
    y_i = test.table[i, ncol(test.table)]
    # produced output for example i
    hat_y_i = g(x_i %*% weights - theta )
    cat(y_i, "\t", hat_y_i, "\n")
  }
}

# define a function to run a simple example
ADALINE.run.simple <- function() {

  # training table
  train.table = matrix(c(0.00, 0,
                         0.05, 0,
                         0.10, 0,
                         0.15, 0,
                         0.20, 0,
                         0.25, 0,
                         0.30, 0,
                         0.35, 0,
                         0.40, 0,
                         0.45, 0,
                         0.50, 0,
                         0.55, 1,
                         0.60, 1,
```

```
                        0.65, 1,
                        0.70, 1,
                        0.75, 1,
                        0.80, 1,
                        0.85, 1,
                        0.90, 1,
                        0.95, 1,
                        1.00, 1),
                      nrow=21,
                      ncol=2,
                      byrow=TRUE)

# test table
# target outputs are shown but non used in testing stage
test.table = matrix(c(0.025, 0,
                       0.075, 0,
                       0.125, 0,
                       0.175, 0,
                       0.225, 0,
                       0.275, 0,
                       0.325, 0,
                       0.375, 0,
                       0.425, 0,
                       0.475, 0,
                       0.525, 1,
                       0.575, 1,
                       0.625, 1,
                       0.675, 1,
                       0.725, 1,
                       0.775, 1,
                       0.825, 1,
                       0.875, 1,
                       0.925, 1,
                       0.975, 1),
                     nrow=20,
                     ncol=2,
                     byrow=TRUE)

# train ADALINE to find weights and theta
training.result = ADALINE.train(train.table)

# test the ADALINE with the final weights and theta
```

```
  ADALINE.test(test.table, training.result$weights,
                    training.result$theta)
  return(training.result)
}
```

The source code for ADALINE.r must be loaded in the R Statistical by typing `source("ADALINE.r")` into the command line and only thereafter the function `ADALINE.run.simple()` can be executed.

The textual output obtained will be similar to Listing 1.2. Notice that the predicted output nearby 0.5 may be wrong, since this is the transition value between the two classes.

Listing 1.2. text output produced by ADALINE.run.simple()

```
> ADALINE.run.simple()
Randomizing weights and theta in range [-0.6, 0.6]

      Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  2
Sum of squared errors =  1
Sum of squared errors =  0
#yi   hat_yi
0       0
0       0
0       0
0       0
0       0
0       0
0       0
```

```
0       0
0       0
0       0
1       0
1       1
1       1
1       1
1       1
1       1
1       1
1       1
1       1
1       1
$weights
[1] 0.8226181

$theta
[1] 0.45
```

It is worth also to visualize how the ADALINE separates the input space. The function `ADALINE.simple.hyperplane.plot()` shown in Listing 1.3 can be executed to plot the hyperplane with the weight and threshold obtained from the network.

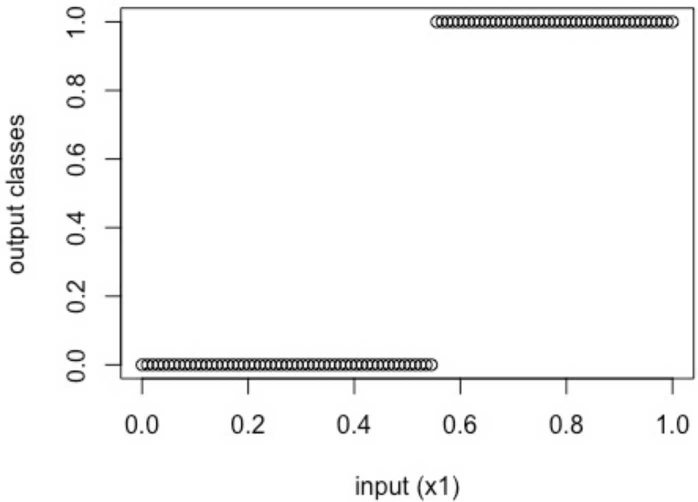
Listing 1.3. Hyperplane plotting

```
# Define the function that plots the hyperplane for the problem
# considering a single input variable x_1
ADALINE.simple.hyperplane.plot <- function(weight, theta,
                   # Variables range.start and range.end define the
                   # interval of values for the single input variable
                                              range.start = 0,
                                              range.end = 1) {
  # Number of variables is 1
  nVars = 1

  # Define the same range for the input variable
  # containing 100 discretized values
  range_of_every_input_variable =
                    seq(range.start, range.end, length = 100)
  x_1 = range_of_every_input_variable
```

```
  # Compute the net for every input value of variable x_1
  all_nets = cbind (x_1, -1) %*% c(weight, theta)

  # Compute the outputs produced by the ADALINE
  # applying the activation function
  hat_y = rep(0, length(all_nets))
  for (i in 1:length(all_nets)) {
    hat_y[i] = g(all_nets[i]) }

  # Construct the hyperplane matrix
  # containing input values of x_1 and the produced output
  hyperplane = cbind(x_1, hat_y)

  # Plot the hyperplane found by the ADALINE
  plot(hyperplane, xlab = "input (x1)", ylab = "output classes")
  return(hyperplane)
}
```

The resulting plot of the hyperplane will be similar to the one illustrated in Figure 1.6.

Figure 1.6. Separation of the input space



The MLP network

The Perceptron and the ADALINE provide accurate results only with classes that are linearly separable. For nonlinear problems additional layers of several neurons are needed in order to build a multilayer perceptron (MLP) architecture, which is the most famous and widely employed ANN. The MLP is composed by the input layer that receives signals from the external environment, at least one hidden layer that extracts patterns and features by processing the data and codifying it and the output layer that produces several final outputs. As shown in Figure 1.7, the signal of each input is propagated layer by layer towards the final layer and always flows in one direction. The number of hidden layers and the corresponding number of neurons depends on many factors, such as the class of the problem considered and the initial values of the training parameters.

The training process of an MLP network is made by using the Backpropagation (BP) algorithm, also known as generalized Delta rule. The special feature of this algorithm is the way in which the error is calculated: the error is computed for the output layer, then it is propagated to the hidden layers and finally to the input layer.

Figure 1.7. MLP architecture with 2 hidden layers

Source: Da Silva et al. (2017)

The error is computed as a function of the network weights and BP algorithm searches the error surface using Gradient Descent for the global minimum. This process is iterated many times and each iteration is formed by two stages: forward activation and backward propagation. The forward activation stage involves

entering one training example into the network in order to obtain a solution to the fed example with the current weights and thresholds. In the backward stage, the responses produced by the network are compared to the desired responses and the difference between ANN and target outputs is used to adjust the weights and thresholds, starting from the output layer, next all the hidden layers and finally to the input layer. These stages are performed repeatedly until the error meets a prespecified stopping criterion.

The weight change can be written as:

$$\Delta w_{ji}^{l} = \eta \cdot \delta_j^l \cdot x_i^{l-1} + \mu \cdot \Delta w_{ji}^{l(previous)} \tag{1.12}$$

where Δw_{ji}^{l} is the incremental change in the weight residing in the interlayer l and connecting node j of the interlayer l with node i of the preceding interlayer $l-1$, x_i^{l-1} is the input coming from interlayer $l-1$ and integrated by node j, η is the learning rate, δ_j^l is the local gradient related to node j in the interlayer l , μ is the momentum coefficient and $\Delta w_{ji}^{l(previous)}$ is the incremental change in the weight residing in the interlayer l and connecting node j of the interlayer l with node i of the preceding interlayer $l-1$ of the previous observation.

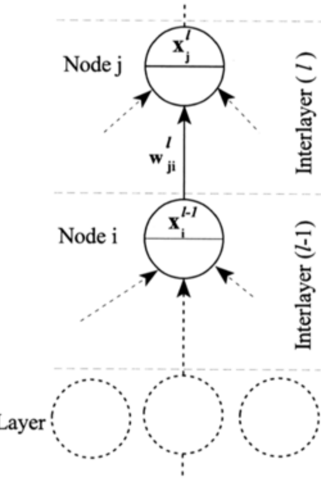
To fully understand the mathematical formulation of Equation 1.11, it is important to reason about the way in which the neurons in one layer get activated in relation to the activation of the neurons in the previous layer, since every neuron in layer l is connected by interlayer connections to all neurons in layer $l-1$. Considering this relationship between the neurons, plus for notation purposes, the interlayer l is defined as the area between two successive layers and it comprises the connection weights inside this area and only the neurons of the upper layer, as can be seen from Figure 1.8. In this way the net produced by neuron j in the interlayer l can be computed according to:

$$\xi_j^l = \sum_{i=1}^{N_{l-1}} w_{ji}^l \cdot x_i^{l-1} \tag{1.13}$$

where ξ_j^l is the net produced by neuron j in the interlayer l, x_i^{l-1} is the signal coming from the ith neuron in interlayer $l-1$ and w_{ji}^l is the weight connecting neuron j to neuron i.

The net is then compared to the threshold by the neuron and the result is used to compute the output by applying the activation function. This output signal is sent to the neurons of the following layer as an input signal and the process is repeated until the signal reaches the output layer[5].

Figure 1.8. Notation and index labelling for the interlayers



Source: Basheer and Hajmeer (2000)

To summarize, in a backpropagation ANN the data is fed unidirectionally without any feedback and the neurons in each layer can be fully or only partially interconnected to the other layers. If the network has enough hidden layers, the learning process of the MLP will be able to approximate any nonlinear function with good accuracy, efficiency and speed. While the mathematical expression of the algorithm is quite complex, it gives an intuitive and natural interpretation of how changes in weights and threshold will affect the network's behaviour and it provides an insight on the impact of each input on the final output. For all those reasons, backpropagation is one the most widely used learning algorithms and MLP is a versatile and flexible model that is used in many applications like data modelling, forecast, classification, image compression, pattern identification and speech recognition. For example, one popular application is the classification of handwritten digits, a task that the MLP performs with better than 98 percent accuracy (Nielsen, 2015).

[5] For further details about the mathematical analysis of the MLP and the backpropagation algorithm see Da Silva et al. (2017) pp. 55-115 and Basheer and Hajmeer (2000)

Coding the MLP in R

MLP networks can be used to solve simple XOR problems that both Perceptron and ADALINE are not able to handle. In this type of problem, two input variables are considered. The neuron will be activated and produce an output equal to 1 only if the two inputs have different values, otherwise the output will be 0. The training sample considered is shown in table 1.2 in which each pair of inputs x_1 and x_2 is attached to the target output.

Table 1.2. Training sample for the XOR problem

Input x_1	Input x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

The code used to solve this problem shown in Listing 1.4 was taken from De Mello and Ponti (2018)[6]. The set of R functions in this Listing contains the MLP sigmoid activation function, a function that builds up the network architecture by specifying the number of neurons in each layer, a function that produces the output after being fed with the inputs, a function that performs the training and adapts weights and thresholds, a function that tests the MLP, a function that produces a plot with the two hyperplanes cutting the input space and a final function responsible for training the XOR problem with the training set presented in Table 1.2. The MLP architecture is composed of two neurons at the input layer corresponding to the input variables x_1 and x_2, two neurons at the hidden layer corresponding to the hyperplanes and one single neuron at the output layer providing the final answer as a 0 or 1.

Listing 1.4 MLP.r implementation

```
# Define the MLP sigmoid activation function
f <- function(net) {
  ret = 1.0 / (1.0 + exp(-net))
```

[6] For more details about the codes, read De Mello and Ponti (2018) pp. 52-72

```
  return(ret)
}

# Define the function to build up the MLP architecture,
# specifying the size of input, hidden and output layers
# with their respective weights and thetas randomly
# initialized in the interval [-1,1]
mlp.architecture <- function(input.layer.size = 2,
                             hidden.layer.size = 2,
                             output.layer.size = 1,
                             f.net=f) {

  # Create a list to contain the layers information
  layers = list()

  # Construct the hidden layer matrix
  # The term "input.layer.size +1" refers to
  # the number of neurons in the input layer (a weight
  # per unit) plus an additional element to define theta
  layers$hidden = matrix(runif(min = -1, max = 1,
                       n = hidden.layer.size * (input.layer.size+1)),
                       nrow = hidden.layer.size,
                       ncol = input.layer.size +1)

  # Construct the output layer matrix
  layers$output = matrix(runif(min = -1, max = 1,
                     n = output.layer.size * (hidden.layer.size +1)),
                     nrow = output.layer.size,
                     ncol = hidden.layer.size +1)

  # Define a list to return:
  # - the number of units at the input, hidden and output layer
  # - layers information (including weights and thetas)
  # - the activation function used
  ret = list()
  ret$input.layer.size = input.layer.size
  ret$hidden.layer.size = hidden.layer.size
  ret$output.layer.size = output.layer.size
  ret$layers = layers
  ret$f.net = f.net

  return(ret)
```

```
}

# Define the function that produces the MLP output
# after providing input values
# The term "architecture" refers to the model
# produced by function mlp.architecture()
# The term "dataset" corresponds to the examples
# used as input to the MLP
# The term "p" is associated to
# the identifier of the current example being forwarded
forward <- function (architecture, dataset, p) {

  # Organize the dataset as input examples x
  x = matrix(dataset[,1:architecture$input.layer.size],
          ncol = architecture$input.layer.size)

# Organize the dataset as expected classes y associated to
# input examples x
  y = matrix(
  dataset[ ,(architecture $input.layer.size +1) : ncol(dataset)],
  nrow = nrow (x))

# Submit the p-th input example to the hidden layer
net_h = architecture$layers$hidden %*% c(as.vector(ts(x[p,])), 1)
f_net_h = architecture$f.net(net_h)

# Organize Hidden layer outputs as inputs for the output layer
net_o = architecture$layers$output %*% c(f_net_h, 1)
f_net_o = architecture$f.net(net_o)

# Define the list of final results produced by the MLP to be returned
ret = list ()
ret$f_net_h = f_net_h
ret$f_net_o = f_net_o
return ( ret )
}

# Define the function responsible for training
# that adapts weights and thetas for every neuron by applying GD
method
backpropagation <- function (architecture, dataset,
                              eta = 0.1, epsilon = 1e-3) {
```

```
  x = matrix(dataset[,1:architecture$input.layer.size],
          ncol = architecture$input.layer.size)
y = matrix(
    dataset[ ,(architecture $input.layer.size +1) : ncol (dataset)],
    nrow = nrow(x))

  cat("Input data ...\n")
  print(x)

  cat("Expected output ...\n")
  print(y)

  cat("Enter to start running ... ")
  readline ()

  squared_error = epsilon * 2

  # Define a loop that will run until the average squared error
  # falls below a certain precision level
  while (squared_error > epsilon) {

    # Initialize the squared error to measure the loss for
    # all examples in the training set
    squared_error = 0

    for (p in 1:nrow(x)) {

      # Applying the input example at row p
      f = forward(architecture, dataset, p)

      # Use the results to adapt weights and thetas
      error = (y[p,] - f$f_net_o)

      # Compute the term "delta" for the output layer
      delta_o = error * f$f_net_o * (1-f$f_net_o)

      # Compute the squared error to be used as a stopping criterion
      # the term "sum(error ^2)" is used becuase the output
      # layer can have more than one neuron
      squared_error = squared_error + sum(error ^2)

      # Compute delta for the hidden layer
```

```
      w_o = architecture$layers$output[,
                 1:architecture$hidden.layer.size]
      delta_h = (f$f_net_h * (1 - f$f_net_h)) *
                 sum(as.vector(delta_o) * as.vector(w_o))

      # Adapt weights and thetas at the output layer
      architecture$layers$output =
        architecture$layers$output + eta * delta_o %*%
        c(f$f_net_h, 1)

      # Adapting weights and thetas at the hidden layer
      architecture$layers$hidden =
        architecture$layers$hidden + eta * delta_h %*% c(x[p,], 1)
    }

    # Find the average (to be used as stopping criterion)
    # by dividing the total squared error by nrow
    # and print it
    squared_error = squared_error/nrow(x)
    cat("Squared error = " , squared_error , "\n")
    }

  # Returning the trained architecture, which can be now executed
  return(architecture)
}

# Define the function to test the MLP
mlp.test <- function(architecture, dataset, debug = T) {

  # Organize the dataset as input examples x
  x = matrix(dataset[,1:architecture$input.layer.size],
             ncol=architecture$input.layer.size)

  # Organize the dataset as expected classes y
  # associated to input examples x
  y = matrix (
    dataset [ ,( architecture $input.layer.size +1) : ncol (dataset
         )],
    nrow = nrow(x))

  cat("Enter to start testing ... ")
  readline()
```

```
  output = NULL

  # For every example at index p
  for (p in 1:nrow(x)) {

    # Apply the input example at row p
    f = forward(architecture, dataset, p)

    # If debug is true, show all information regarding classification
    if (debug) {
      cat("Input pattern = ", as.vector(x[p,]) ,
          " Expected = ", as.vector(y[p,]) ,
          " Predicted = ", as.vector(f$f_net_o), "\n")
    }

    # Concatenate all output values as rows in a matrix
    # to be able to check them
    output = rbind(output , as.vector(f$f_net_o))

  }

  # Return results
  return(output)
}

# Define the function to produce a discrete hyperplane to
# shatter the input space of examples
discretize.hyperplane <- function(img, range = c(0.45, 0.55)
                                  ){
  ids_negative = which(img < range [1])
  ids_positive = which(img > range[2])
  ids_hyperplane = which(img >= range[1] & img <= range[2])

  img[ ids_negative ] = 0
  img[ids_positive] = 1
  img[ ids_hyperplane ] = 0.5
  img
}

# Define the function to train and test a XOR problem
xor.test <- function(eta = 0.1, epsilon = 1e-3) {
```

```
  # Load the dataset "xor.dat"
  dataset = as.matrix(read.table("xor.dat"))

  # Build up the MLP architecture with random weights and thetas.
  # There are 2 units at the input layer corresponding to number
  # of input variables, 2 units at the hidden layer corresponding
  # to the hyperplanes and 1 unit at the output layer to provide
  # the answer as values in range [0 ,1]
  model = mlp.architecture(input.layer.size = 2,
                           hidden.layer.size = 2,
                           output.layer.size = 1,
                           f.net = f)

  # Train the architecture "model" to build up the "trained.model"
  trained.model = backpropagation(model, dataset, eta = eta,
                                  epsilon = epsilon)

  # Test the "trained.model" using the same XOR dataset
  mlp.test(trained.model, dataset)

  # Build up hyperplanes to plot
  x = seq (-0.1 ,1.1 , length =100)
  hyperplane_1 = outer(x,x,
                       function(x,y) { cbind(x,y,1) %*%
                           trained.model$layers$hidden[1,] } )
  hyperplane_2 = outer(x,x,
                       function(x,y) { cbind(x,y,1) %*%
                           trained.model$layers$hidden[2,] } )
  cat("Press enter to plot both hyperplanes ... ")
  readline ()

  # Plot the hyperplanes built at the hidden layer
  filled.contour(discretize.hyperplane(hyperplane_1) +
                 discretize.hyperplane(hyperplane_2))
}
```

The source code of MLP.r has to be loaded in the R Statistical Software typing `source("MLP.r")`, the working directory must be set to the folder containing the training data table saved as `xor.dat` and only then the function `xor.test()` can be run. The textual output in Listing 1.5 shows the input examples and the related

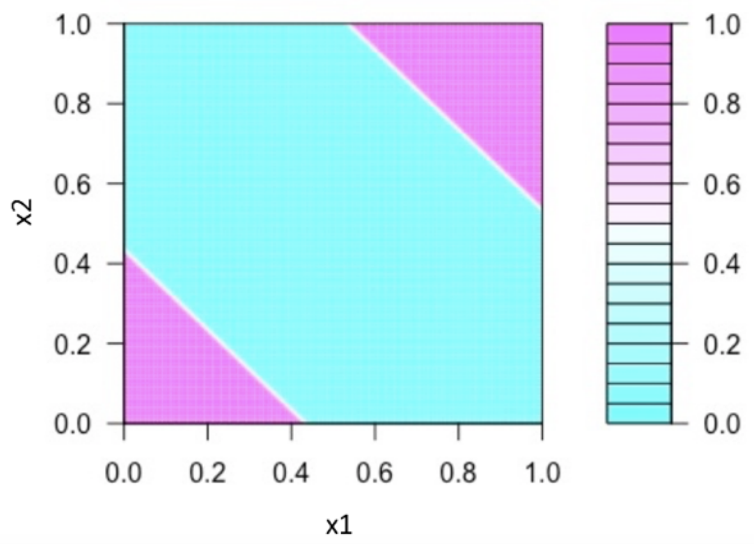
outputs. Then by typing "enter", training is performed and the average squared error is shown for each iteration, until convergence is reached. By typing "enter" again, the results of the classification task are shown. The predicted outputs are not exactly the same as the target ones, yet they are sufficiently similar. One could set a smaller precision rate (epsilon) to obtain a better approximation, depending on degree of precision that is needed. However, this choice will slow down learning.

Listing 1.5. Textual output of the MLP

```
> xor.test()
Input data ...
     [,1] [,2]
[1,]    0    0
[2,]    0    1
[3,]    1    0
[4,]    1    1
Expected output ...
     [,1]
[1,]    0
[2,]    1
[3,]    1
[4,]    0
Enter to start running ...
.
.
.
Squared error =  0.001000326
Squared error =  0.001000248
Squared error =  0.00100017
Squared error =  0.001000092
Squared error =  0.001000014
Squared error =  0.0009999364
Enter to start testing ...

Input pattern =  0 0  Expected =  0  Predicted =  0.02902642
Input pattern =  0 1  Expected =  1  Predicted =  0.9668781
Input pattern =  1 0  Expected =  1  Predicted =  0.9668705
Input pattern =  1 1  Expected =  0  Predicted =  0.03100915
Press enter to plot both hyperplanes ...
```

Typing “enter” once again a plot of the two inferred hyperplanes will be produced as shown in figure 1.9. The light blue region between the two hyperplanes contains the input couples that produce the output 0, while the pink regions outside the hyperplanes contain the input couples attached to the output 1.

Figure 1.9. Plot of the hyperplanes cutting the input space



1.7 Other popular ANNs

A vast number of artificial neural networks have been developed for many different applications in addition to the Perceptron, ADALINE and MLP. In this section some of the most frequently used networks are summarized.

The Hopfield network is a symmetric fully connected recurrent network, with an input and an output layer. It acts as a non-linear associative memory and is able to classify incomplete or noisy input signals by using its internally stored characteristics. However, it is appropriate only for binary inputs. The weights connecting two neurons are set equal to the product of their inputs and learning is done by implementing an energy function.

The adaptive resonance theory (ART) network is composed by two fully interconnected layers, a layer receiving input signals and an output layer. It is trained with unsupervised learning and have two sets of weights: the feedforward weights are used to select the winning output neuron and keep long-term memory,

while the feedback weights test the vigilance and keep the short-term memory. This type of training allows the system to store the classified characteristics and compare them with each new example in order to decide if it is dissimilar enough to store it as a new characteristic or if it matches the existing ones.

Kohonen networks are composed by two layers that transform high-dimensional inputs into data of lower order. They are trained with unsupervised learning and group data into clusters. In practice they are used for compressing data and recognizing patterns.

Counterpropagation networks are used to create a self-organizing lookup table, by hybrid learning. Unsupervised learning is implemented to create a Kohonen map of inputs in order to group data into clusters. At the same time, supervised learning is applied to this map in order to associate an output to each point on the map. After the training phase, new examples are classified simulating the lookup table.

Radial basis function (RBF) networks are MLP error-backpropagation networks with three layers. The hidden layer function is to cluster the inputs. The activation function used is the Gaussian kernel and belong to the radial basis functions. In practice they are used for function approximation, classification and time series prediction (Basheer and Hajmeer,2000).

1.8 ANNs compared to statistics and expert systems

It is worth to compare ANNs to statistics and expert systems (ES), given that they are different on many aspects and yet they share plenty of similarities. Basheer and Hajmeer (2000) and Anderson and McNeill (1992) discussed this matter in detail.

Considering the comparison with statistics, Basheer and Hajmeer (2000) noted that some models, such as feedforward network with no hidden layers, are a generalization of statistical models, other models like Hebbian learning are related to statistical modelling and others, such as Kohonen networks, have no similarities with statistical models. Some researchers believe that ANNs are a type of nonlinear statistical regression or a generalization, others view them as superior tools. For

classical regressions, the mathematical equation and independent variables must be known or assumed a priori and the function relating the variables is a set of linear operators. The internal structure of an ANN, on the other hand, is represented by a complex function combining a large number of nonlinear functions.

The most important characteristic is that ANNs have a higher predictive accuracy than statistical regression and the accuracy increases as the dimensionality and nonlinearity of the problem increases.

Depending on the type of problem that has to be solved, one model will be preferred to the other. When modelling data with only few dimensions or approximating simple functions, statistical techniques should be chosen. ANNs will be used instead when a high accuracy is needed or when the problem has high dimensionality and complexity.

Expert systems are computer programs that try to mimic the logical reasoning process of the human brain and rely on rules, concepts and calculations. An ES is composed by an inference engine (a generic component) and a knowledge base, containing the information related to a specific problem and allowing the programmer to define a set of logical (if-then) rules. The engine draws conclusions and provides a solution based on the rules (Anderson and McNeill, 1992).

ESs process information in a sequential manner and suffer from some limitations due to their high sensitivity to incomplete or noisy data. This is because expert systems do not learn by themselves, instead they follow the rules that were manually defined by the programmer and are not able to deal with incomplete or noisy data. They need to be fed with accurate and consistent data with respect to the rules set. In addition, when the complexity of the system increases, it will become too slow and ask for too many computing resources. Moreover, not all kind of knowledge can be expressed in terms of a set of logical rules.

The choice between ANNs and expert systems also depends on the type of problem. Expert systems can be used for problems in which data and theory are not adequate, by coding the information with rules of thumb. Conversely, ANNs are preferred for solving problems where plenty of data is available but no clear theory can be formulated. For further details see Basheer and Hajmeer (2000).

CHAPTER 2
The development of an ANN

The present chapter is focused on presenting and discussing the methodology and the precise steps that have to be followed in order to develop a new ANN project. When developing an ANN, the modeler has to account for many issues relating to the dataset, the architecture and training of the model. A summary of the issues and possible solutions, which are well described by Basheer and Hajmeer (2000), will be presented in the second section of the chapter.

2.1 Steps in an ANN development

In order to develop an ANN project from scratch, seven steps need to be taken, as shown in Figure 2.1. The first step requires choosing the appropriate model outputs and a set of potential input variables from the available data. In this phase, the problem has to be defined and formulated. It is important to understand the causal nexus between the different variables and to assess the possible benefits over other techniques, such as statistics and expert systems. The starting point is to collect all possible variables that could help to explain the problem in question and then choose the inputs that will be used in the model, based on prior knowledge or availability of data. The resulting dataset forms the selected (unprocessed) dataset.

The second step consists in pre-processing all the variables from the selected unprocessed dataset, so that they can be fed into the network in an appropriate form. The resulting dataset forms the selected (processed) dataset.

In the third step, the vector of appropriate inputs is selected. This is an important choice in the ANN development process, because the inclusion of too many correlated inputs will increase the training time and the overfitting

probability, while the exclusion of important variables will lead to a model that is not able to fully explain the problem.

The fourth step relates to the division of the dataset into two subsets (training and test), which will be used for the model calibration and the performance evaluation phases. For networks that are prone to overfitting, usually cross-validation is used as the stopping criterion for the model structure selection and the training set is further randomly partitioned into an in-sample set and an out-of-sample (or validation) set of chosen length. The in-sample set is used to train different candidate models that have diverse architectures in order to estimate the weights. The validation set is employed to decide which is the best parameters configuration (e.g. number of nodes and number of hidden layers) for each model, in order to prevent overfitting. The test set is an independent dataset, which is used to evaluate the performance and generalization capability of the best models. There is a lot of confusion about the terminology used for the three subsets. Many industrials and academics invert the meaning of the validation and test sets. The same confusion arises also for the division into three sets rather than two. In this case, the in-sample set is called training set, while the validation set and the test set preserve their meaning.

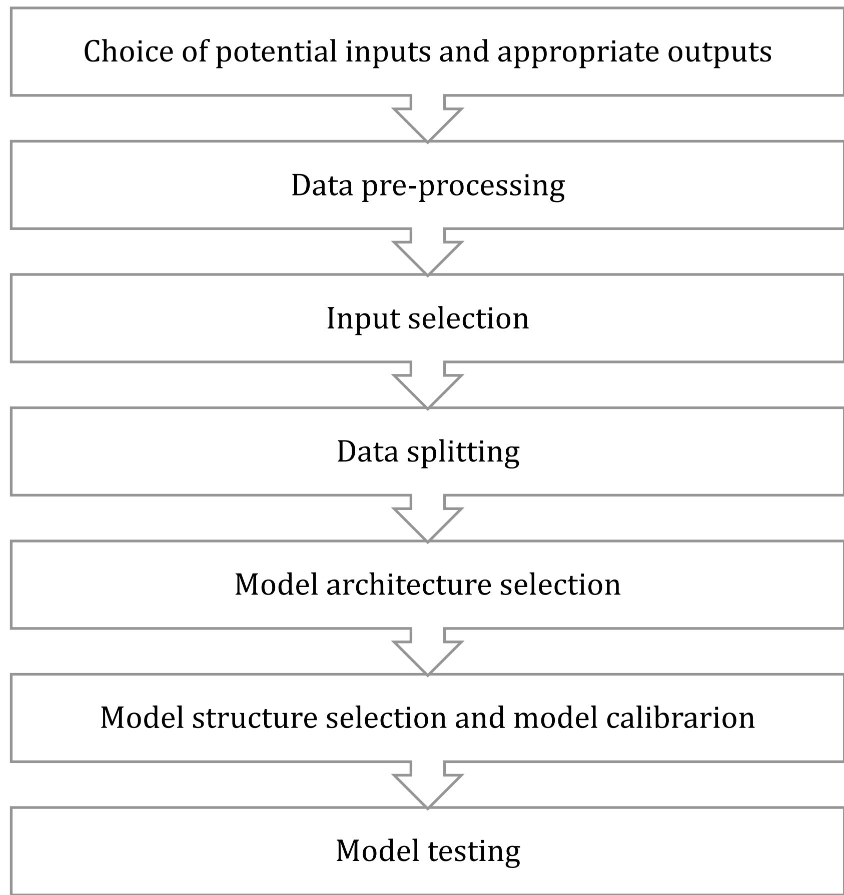
The fifth step requires choosing a set of model architectures that will be developed, such as feedforward or recurrent. This will determine the overall structure and the information flow of the model.

The sixth step consists in selecting the best model structure and calibrating each model with the training set. The biggest issue in this phase is to select the right size of a network (layers and nodes), the learning rate, the number of training epochs and an adequate precision rate. All these elements will affect the design and the performance of the network and it is recommended to try and design a set of possible networks with different configurations in order to reach an optimal selection of the parameters. The selected models are calibrated using cross-validation in order to obtain the optimal parameters for each model.

The seventh step relates to model testing (or performance evaluation). The calibrated models need to be tested using an independent data set in order to assess their generalization capability and accuracy when presented with unseen examples. In this phase it is possible also to compare the performance of the ANN to the other

techniques cited in the first step. For further details see Basheer and Hajmeer (2000), Maier et al. (2010) and Wu et al. (2014).

Figure 2.1. Steps in developing an ANN project



2.2 Related issues and possible solutions

When developing an ANN based model, a series of issues pertaining to the database creation and processing, weight initialization, choice of the learning rate, momentum coefficient, convergence criteria, training epochs, size of hidden layers and many others have to be addressed before staring the training process.

Database size and partitioning

An ANN capability to generalize will highly depend on the size of the database, given that they are utilized as interpolators with unseen examples and the size must be sufficiently large to enclose all the possible deviations inside the problem domain.

The training subset contains all the data relevant to the problem and it is employed during the training step in order to update weights and thresholds. The validation subset contains different data from the training subset, but still in the domain boundaries of the problem. This set is used in the learning process for choosing the best parameter configuration. The validation subset includes data different from the other two subsets and is employed to select the optimal structure and check its accuracy and generalization capability.

No precise rules govern the division into the three subsets, however, there are some rules of thumb followed by many researchers. The training subset could be set equal to at least the number of weights times the inverse of the minimum desired error. Other researchers propose to divide the dataset into 65% for training, between 20% and 30% for validation and around 10% for testing. From researchers' experience this division gives good generalization capability.

Data pre-processing, balancing and enhancement

The dataset usually cannot be directly inputted into the network but has to be pre-processed in order to accelerate convergence. Some of the most used techniques require removing noise, reducing the dimensionality of the inputs, transforming data, treating non-normally distributed data, inspecting data and deleting outliers.

For classification problems, it is essential that the dataset is well balanced, particularly the training subset should be divided evenly between the different classes, otherwise it will be biased towards a specific class that is overrepresented. Possible solutions are to remove completely or diminish the examples of overrepresented classes, alternatively to add some examples to the classes that are inadequately represented. A second solution is to duplicate the underrepresented input plus output couples and add some random noise to the inputs.

Another issue relates to datasets of small size, that are difficult to subdivide into the three subsets. If possible, new data should be added, otherwise it is still possible to enrich data by introducing random noise to existent examples to produce new ones. This technique will increase model robustness and diminish prediction errors. If enriching data is not feasible, other techniques such as leave-one-out method, grouped cross-validation and bootstrap can be applied.

Data normalization or scaling

Another processing that the data has to undergo is the normalization or scaling within a uniform range. This is important in order to prevent large numbers from overriding small numbers and to avoid premature saturation of the hidden layers with a consequent slowdown in the learning rate, considering that real world numbers are not uniform and are distributed on a quite differing range.

A possible normalization approach, that is recommended by Basheer and Hajmeer (2000), consists in scaling input and output variables z_i in the interval $[\lambda_1,\ \lambda_2]$ that corresponds to the range of the activation function:

$$x_i = \lambda_1 + (\lambda_2 - \lambda_1) \cdot \left(\frac{z_i - z_i^{min}}{z_i^{max} - z_i^{min}} \right) \tag{2.1}$$

where x_i is the scaled value of z_i and z_i^{max} and z_i^{min} are the maximum and minimum values of z_i in the dataset.

For the sigmoid transfer function, it is recommended to choose a slightly offset interval, such as $[0.1, 0.9]$ to avoid premature saturation and slowdown in the learning process. In case that the range of original data is particularly large, the logarithm can be taken before the scaling procedure.

Data representation

The representation of inputs and outputs can be discrete, continuous or a mixture of the two. For classification problems where binary inputs and outputs can be defined, the particular representation chosen will determine the dimensionality of the vector of inputs or outputs. For example, if the network receives two input signals and the result generated by the transfer function for each input is one of the

four levels of activation such as low, medium, high and very high, then a two digit representation, that transforms the input vector into four inputs, can be given to each input such as 00, 01, 10 and 11. As an alternative, it can be used a four digit representation such as 0001, 0010, 0100 and 1000 in which the position of 1 defines the level of activation. In this case the input vector is converted into eight inputs: four binary numbers for the first input and four also for the second input.

Many other binary representations can be chosen and even continuous classes can be transformed into binary numbers by simply dividing the range into n intervals and assigning to each interval a class. The advantage of binary data relates to the derivation of rules for trained networks.

Weights and thresholds initialization

The network initialization of weights and thresholds was found by several studies to affect the speed of convergence. This mainly depends on the positioning of the weight vector on the error surface and in case it falls in a flat region, convergence will slow down or even stop due to a premature saturation of the neurons.

In order to circumvent this issue, each weight and threshold is usually initialized by randomly drawing a number from a small range of a normal or uniform distribution with a zero mean. Other distributions may also be considered. However, there is also the risk of choosing a range that is too small and consequently to slow down the learning process.

There are several possible approaches for the choice of initial values. Some suggest choosing random weights in a range of $[-0.3, +0.3]$. Others suggest a neuron by neuron assignment in which, for each neuron, uniformly sampled numbers are taken from the interval $\left[-\frac{r}{N_j}, +\frac{r}{N_j}\right]$, where r is a real number subject to the transfer function and N_j is the number of connections delivering signals to neuron j. Another alternative is to consider an interlayer, rather than a single neuron. In this case weights are sampled from the interval $\left[-3M^{-\frac{1}{2}}, \quad +3M^{-\frac{1}{2}}\right]$, where M is the number of weights to be assigned for the interlayer.

Learning rate and momentum coefficient

Two fundamental parameters in backpropagation neural networks, namely the learning rate η and the momentum coefficient μ, must also be set carefully to reach convergence.

The learning rate defines the width of the change in the weight vector in order to reach the minimum error during training. If η is small, the steps taken are tiny and the learning process will be slow. If η is large, training will speed up, however the risk of over-shooting the minimum when it is in its proximity increases and convergence may never be reached as the weight vector continues to oscillate on the error surface.

Some authors suggest using a small constant learning rate between $[0.3, 0.6]$ or $[0, 1]$, while others advise an adaptive rate that takes larger steps at the beginning of the training and smaller steps when the search moves closer to the minimum.

The purpose of the momentum coefficient is to help the search not getting stuck in a local minimum and to reduce the possibility of search instability. When a small η tends to slow down learning, the term μ speeds up the updating process of the weights. Although, a too high μ suffers from the risk of overshooting as does a high η and a too low μ will lead to slow training.

Different solutions have been proposed in regard to the choice of the momentum. Recommended values range from 0 to 1 or as an alternative momentum and learning rate are constrained together to the sum $\mu + \eta = 1$. Other approaches consist in decreasing μ as learning speeds up or changing μ based on the error gradient information. Researchers usually start with one of those recommended values and then, depending on the problem considered, they will find the optimal parameters by a trial and error procedure.

Convergence criteria

The error function ρ represents the deviations of network outputs from the target values and the training process moves forward until the error reaches a required precision (ε or δ). For the weight adjustment process, it is essential to define a proper error function and a precision rate.

The error can be measured in many different ways (Wu et al., 2014). The main categories are:

- squared errors, which are based on the square of the difference between the target and predicted outputs. They include the mean square error (MSE), the root mean square error (RMSE) and the sum of squared errors (SSE);
- absolute errors, which are based on the absolute difference between the target and predicted outputs. They include the mean absolute error and the sum of absolute errors;
- relative errors, which indicate the magnitude of the differences between target and predicted outputs relative to the target outputs. They include the average absolute relative error, the mean percentage error and the relative bias;
- information criteria, such as AIC (Akaike information criterion) or BIC (Bayesian information criterion).

The most common way to measure the error function is through the sum of squared errors (SSE) and it can be computed both for the in-sample and the validation subsets or for the training sample, depending on the initial division into subsets. It can be estimated according to:

$$SSE = \sum_{j=1}^{N}\sum_{i=1}^{M}(d_{ji} - y_{ji})^2 \quad (2.2)$$

where d_{ji} is the target solution of the *i*th output neuron on the *j*th example, y_{ji} is the ANN output produced by *i*th neuron for the *j*th example, N is the number of examples and M is number of output nodes.

There are different convergence criteria that can be applied to stop the training process and three examples are presented next. In the first two, overfitting is not viewed as a problem and thus the training set does not have to be split up into two sets. It is sufficient to train and test the network. For the third case, overfitting is a serious issue and cross-validation is needed to find the optimal network structure.

For models such as the Perceptron, learning stops when the difference between the current and the previous error reaches the required precision. The

error belongs to the absolute error category and can be computed for example as the mean absolute error.

For models like the ADALINE, the objective is to minimize the squared error. Learning will depend on the gradient of the squared error function and convergence is reached if the difference between the current and the previous mean squared error becomes lower than the precision rate according to $\left|\bar{E}(\boldsymbol{w}^{current}) - \bar{E}\left(\boldsymbol{w}^{previous}\right)\right| \leq \varepsilon$, as was shown in the previous chapter.

Models with more layers and nodes have numerous parameters, thus the risk of overfitting is very high. It is not easy to find the optimal structure of the model and for this reason, it can be applied the cross-validation of the error function in order to find the optimal number of hidden nodes or the number of training cycles. This criterion is more reliable and prevents overfitting, but it is computationally complex and necessitates a large amount of data.

The training set is divided into the in-sample and the validation sets. The cross-validation method involves training the network on the in-sample set in order to minimize the error function and to find the optimal weights according to the stopping rule chosen, as it is usually done for the Perceptron and the ADALINE. The validation data is then fed into the network having the optimal weights and the validation error is computed. This procedure is repeated by increasing each time the number of hidden nodes or the number of training cycles. The training error and the validation error are monitored together for each addiction and the optimal architecture chosen will be the one right before the beginning of the increase in the error curve for the validation subset. Adding any other nodes or training cycles, will only lead to overfitting.

The idea behind the cross-validation is shown in Figure 2.2. For the training subset the error decrease indefinitely when the number of hidden nodes or the training cycles are increased. The initial decline in error comes from learning, however, the further reduction is due to memorization or overtraining rather than generalization when the process is trained for too many cycles and overfitting when too many hidden nodes are present. The error of the validation subset on the other hand initially decreases and then increases when the network loses its ability to generalize.

Figure 2.2. Cross-validation to select the optimal architecture

Source: Basheer and Hajmeer (2000)

For classification problems, in which the output is a discrete value, a different error metric is used. Usually the convergence criterion depends on the proportion of examples classified in a correct and incorrect way, called a hit-or-miss rate.

Training modes

The update of weights and thresholds can be performed in two different ways by a network: example by example training (EET) and batch training (BT).

In the first type of training, the weights and thresholds are updated every time that an example is fed into the network. The chosen learning algorithm is applied to the first example until the desired error is obtained and this procedure is repeated for one example at a time until the last training example has been learned. The main advantage of this mode is that it requires only a small storage for saving the weights vector. It is also efficient in the stochastic search of the optimal solution and does not get stuck on a local minimum. The major problem, however, is the dependence of the final result on the first example, which could be misleading and lead the search in the wrong direction.

In batch training, on the other hand, all training examples are fed together into the network and the weights are updated afterwards. The error is averaged across all the examples and only then the learning algorithm can be applied to find the minimum error. This procedure is iterated many times, in which all the examples

are presented again to the network in a random way for each epoch. With respect to EET, batch training provides a better estimate of the gradient of the error, which allows to compute a precise measure for the change in weights. The disadvantage is the large storage requirement for the weights and the higher risk to get stuck in local minima and it requires a much longer training time.

The choice between the training modes will depend on the type of problem that is addressed, the amount of data and the storage capacity.

Hidden layer size

The most crucial task in designing an ANN system is to assess the proper number of hidden layers and hidden nodes in each layer. For input and output layers, the number of nodes is defined by the input and output variables specified for the problem in question. Yet, there is no clue about hidden layers' size and number. Researchers usually employ rules of thumbs or a trial and error procedure.

If an ANN has too few hidden nodes, it will not be able to distinguish complex patterns and if, on the other hand, it has too many nodes, it will not be able to distinguish noisy data leading to poor generalization. Another issue to consider is that learning slows down with many hidden nodes and training can become extremely time consuming.

The optimal size of the hidden layers can be related to the inputs and outputs number, the size of the training set or to the nonlinearity issue. Many rules of thumbs relating the number of hidden nodes N_{HN} to the number of input N_{INP} and output N_{OUT} nodes exist. The number of hidden nodes can be computed for example as $N_{HN} \approx \sqrt{N_{INP} \cdot N_{OUT}}$, where the number of hidden nodes is approximated to the closest integer to the result of the square root or it can be determined from $N_{HN} \leq N_{INP} + 1$. Another approach is to relate N_{HN} indirectly to the size of the training set N_{TRN} through the total number of weights N_W according to $\frac{N_W}{N_{OUT}} \leq N_{TRN} \leq \frac{N_W}{N_{OUT}} \log_2\left(\frac{N_W}{N_{OUT}}\right)$ and to choose N_{HN} in order to respect this inequality.

For problems with complex nonlinearities, however, those rules of thumb are not enough and typically a trial and error procedure must be followed.

This section presented several parameters that have to be chosen carefully when designing an ANN system. Most of them are selected through a trial and error procedure and mainly depend on the type of the problem and its degree of nonlinearity. Table 2.1 summarizes the main parameters seen in this chapter together with the possible issues coming from the choice of a too small or too large value.

Table 2.1. Effect of extreme values of system parameters on convergence and generalization capability

System parameter	If too large (high)	If too small (low)
Number of hidden nodes (N_{HN})	Overfitting and bad generalization	Underfitting and inability to find the underlying rules
Learning rate (η)	Overshooting of the optimal solution	Slow training
Momentum coefficient (μ)	Increased risk of overshooting the optimal minimum	Slow learning
Number of training cycles	Bad generalization ability (the system memorizes data)	Inability to represent the data characteristics
Size of training set (N_{TRN})	Good recalling and generalization capability	Inability to fully explain the problem and limited generalization
Size of the test set (N_{TST})	Generalization capability can be confirmed	Generalization capability cannot be adequately confirmed

Source: Basheer and Hajmeer (2000)

CHAPTER 3
Deep learning and deep neural networks

The present chapter is devoted to analysing deep learning and the great potential of deep neural networks. The main characteristics that differentiate deep architectures from shallow ones will be discussed. The following sections will reason on why deep neural networks are hard to train and how researchers addressed this issue. New architectures, such as convolutional neural networks and long short-term memory networks, were created and new training techniques were implemented.

3.1 Deep learning characteristics and history

Deep learning is a machine learning technique that makes use of deep neural networks and is based on learning complex data representations by examples. Deep neural networks (DDNs) are networks characterised by a multi-layered architecture and comprise two or more hidden layers. They differ from shallow networks, which contain only a single hidden layer.

Multilayer ANNs are able to discover complex and nonlinear functional mappings, if they are provided with enough data and computational resources. The advantage in their use is the reduction of large amounts of manual work together with an outstanding performance when they are fed with unstructured and high-dimensional data. Considering the complexity of a deep network, it can be defined as black box system, since the user has no understanding of the internal working of the model, which can only be viewed in terms of its inputs and outputs. Black box systems are different from white box systems in which the internal structure of the model is known (Sengupta et al., 2020).

DNNs are built of multiple processing layers that are able to learn data representations (how the data is structured) by using multiple levels of abstraction that describe the degree of complexity. This is an important characteristic for deep learning techniques: representation learning allows the system to be inputted with raw data and to discover automatically (so not designed by engineers) the proper internal representation or a feature vector from which the network can detect or classify input patterns. Starting from raw inputs, each hidden layer in the network transforms the representation into a more complex abstract level and after all the transformations, the network will be able to learn highly complex functions.

Higher layers of representation are essential for classification tasks, since they increase both selectivity and invariance of the representation. Aspects that are important for discrimination are amplified (selectivity), whereas irrelevant variations are suppressed (invariance).

To better understand the abstraction mechanism, consider for example the application of object identification in images, which involves visual pattern recognition. An image is formed by an array of pixels and the network is responsible for breaking down a complicated task, such as detecting if the image shows a human face, into very simple tasks that can be answered at the level of each pixel. For instance, the first layer could learn to recognize edges at particular orientations and locations in the image, the second layer could recognize particular arrangements of edges such as triangles or rectangles, the third layer could detect even more complex combinations of shapes and so on. Later layers will thus build up a complex hierarchy of abstract concepts.

It is important to notice that shallow networks are not able to distinguish between relevant and irrelevant variations of inputs on their own and instead the feature extractors need to be hand designed by the programmer. Variations in background, surrounding objects, orientation, pose or illumination are irrelevant in an image. On the other hand, some particular minute details are the key to distinguish between two similar objects, such as images of two dog breeds.

Deep neural networks implement extremely intricate functions and are able to solve problems of image recognition with a much higher accuracy than shallow architectures. For further details see LeCun et al. (2015) and Nielsen (2015).

Multilayer ANNs, such as the MLP, were known for many years. However, they have gained substantial attention of academics and professionals only in the recent years. Interest in deep neural networks was revived in 2006 when Hinton proposed a new architecture called deep belief network (DBN) and a new training method called layer-wise-greedy-learning. This method involves pre-training an unlabelled dataset with unsupervised learning before the subsequent layer-by-layer training in order to extract the relevant features. The features extracted are then exported to the next layer, all the samples are labelled and the network is fine-tuned with the labelled data using the standard backpropagation method to further adjust the weights. Using this approach, the dimensionality of the dataset can be greatly reduced and a compact representation is obtained. The main advantages are the mitigation of the overfitting problem, a better reach of local minima and faster convergence (Liu et al., 2017).

In the past decade the research on deep learning gained a great deal of attention. Sengupta et al. (2020) provide s summary of the factors that supported their rise in popularity:

- Recent rise in the availability of large labelled data sets;
- Advent of fast graphics processing units (GPUs), progress in parallel computing capabilities and multi-core implementations;
- Creation of software platforms that permit a smooth integration of the architectures into a GPU computing framework in order to decrease the complexity of environment setup;
- Introduction of better regularization techniques to avoid overfitting and to improve performance, such as L1 and L2 regularization, dropout, batch normalization, data augmentation and early stopping. These techniques will be analysed in detail in section 3.6;
- Implementation of robust optimization algorithms that provide near-optimal solutions, such as stochastic gradient descent.

Deep learning approaches have been applied in many different fields over the years. They are suitable also for analysing big data and are commonly used in computer vision, pattern recognition, speech recognition, natural language processing and recommendation systems.

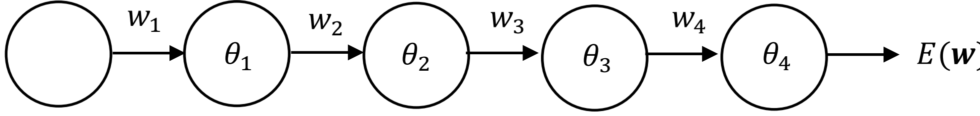
3.2 Reasons why DNNs are difficult to train

Deep neural networks were at first trained using the backpropagation algorithm, however they did not perform much better than shallow architectures and in some cases, they performed even worse. This is a strange result, since adding additional layers should allow the network to learn more complex functions and achieve a greater level of abstraction. The problem is that the backpropagation algorithm is not able to find the optimal weights and biases, when it is used for deep architectures.

In a deep network, different layers learn at extremely different speeds. When later layers in a network learn faster than early layers, the early layers will often get stuck during training and will learn almost nothing. On the other hand, when the early layers learn faster, the later layer will get stuck. This is due to the intrinsic instability that is associated to backpropagation learning by gradient descent in multilayer networks. The fundamental problem (namely the unstable gradient problem) is that the gradient in early layers is the product of terms from all the later layers. All the layers could learn at the same speed only if all those products of terms balanced out. It is very unlikely that this balancing will occur by chance and for this reason there is an intrinsic instability.

Consider the simplest deep neural network having three hidden layers and one single neuron in each layer, as shown in figure 3.1. Here w_i are the weights, θ_i are the thresholds and $E(\boldsymbol{w})$ is the loss function.

Figure 3.1. Simple network with 3 hidden layers and 1 neuron per layer



As an example, the explicit expression for the gradient of the threshold associated to the first hidden neuron can be written as:

$$\frac{\partial E(\boldsymbol{w})}{\partial \theta_1} = g'(u_1) \cdot w_2 \cdot g'(u_2) \cdot w_3 \cdot g'(u_3) \cdot w_4 \cdot g'(u_4) \cdot \frac{\partial E(\boldsymbol{w})}{\partial y} \quad (3.1)$$

where $g'(u_i)$ is derivative of the sigmoid activation function, u_i is the weighted input to neuron i, y is the output produced by the network and $\frac{\partial E(\boldsymbol{w})}{\partial y}$ is the loss function at the end. This expression is a product of terms in the form $w_i \cdot g'(u_i)$.

The vanishing gradient problem

The vanishing gradient problem arises when later layers in a network learn faster than early layers. In this case the gradient tends to get smaller as the algorithm moves backward from the output layer through the hidden layer, meaning that the adjustment in weights in early layers is much smaller and the search moves slowly towards the minimum.

Consider again the gradient associated with the first hidden neuron defined in equation 3.1. The derivative of the sigmoid function reaches its maximum at $g'(0) = 1/4$. If weights are randomly initialised in a range between [-1,1] using the standard approach, the terms $w_i \cdot g'(u_i)$ will usually satisfy $|w_i \cdot g'(u_i)| < 1/4$ and the product of many such terms will tend to rapidly decrease. The more are the hidden layers, the smaller this product will be. For later thresholds the gradient shares some terms with respect to the first threshold, but the expression is shorter. Consider for example the gradient for the threshold in the third layer:

$$\frac{\partial E(\boldsymbol{w})}{\partial \theta_3} = g'(u_3) \cdot w_4 \cdot g'(u_4) \cdot \frac{\partial E(\boldsymbol{w})}{\partial y} \quad (3.2)$$

In this case, having less products, the gradient will not be as small as the first one and for this reason learning will be faster for later layers. This is the fundamental reason for which the vanishing gradient problem occurs and it is intrinsic to the backpropagation algorithm and the random initialization of the weights.

It is possible to visualize the effects of this problem on the speed of learning during the training phase of a network. Figure 3.2 shows the learning speed of three different networks having two, three and four hidden layers. The learning speed change was recorded over 500 epochs of training. For the network with two layers, learning starts at different speeds and the speed drops very quickly before

recovering for both the layers. It can be clearly seen that the first hidden layer learns far more slowly than the second layer during all the 500 epochs of training. A similar behaviour occurs for the other two networks and early hidden layers learn much slower than later layers.

Figure 3.2. Learning speed of networks with 2,3 and 4 hidden layers

Source: Nielsen (2015)

The exploding gradient problem

The exploding gradient problem arises when the early hidden layers learn faster than the later hidden layers. The gradient in this case will grow dramatically as the algorithm moves backward through the layers. Consider again the gradient in equation 3.1: if the weights grow during training and become large enough, it is

possible that the terms $w_i \cdot g'(u_i)$ in the product will no longer satisfy $|w_i \cdot g'(u_i)| < 1/4$ and will get larger than 1. This will lead to an exploding gradient.

Deep neural networks are hard to train not only because of the instability of the gradient-based learning. Other factors can also have a significant impact:

- the choice of the activation function should be done by finding alternatives to sigmoid activation functions, which cause problems and slow down training;
- the weights should be initialized with a different approach from the standard random initialization;
- the learning by gradient descent should be implemented by using a different learning algorithm, such as the stochastic gradient based learning algorithm;
- the network architecture and the other parameters should also be chosen carefully[7].

The stochastic gradient descent (SGD) is a popular optimization algorithm used for training deep neural networks. During the training phase the network is fed with a small batch of examples for which it computes the outputs and the errors. The average gradient is computed for those examples and the weights are adjusted accordingly. This process is repeated for many small sets of examples taken randomly from the training set until the average of the objective function stops decreasing. The result is a noisy estimate of the average gradient over all examples given by each small batch. SGD is quicker to converge than the classic backpropagation algorithm since it reduces the computational complexity. The main disadvantage, however, is that it is difficult to find the global minimum when having noisy estimates.

During the recent years, researchers proposed different architectures and training techniques that address the difficulty of training in deep neural networks. In section 3.3 and 3.4 two well-known architectures, namely convolutional neural networks

[7] For further details about the unstable gradient problem and the other issues related to the training of deep neural networks, see Nielsen (2015) chapter 5 available at http://neuralnetworksanddeeplearning.com/chap5.html

and long short-term memory networks, will be discussed. Section 3.6 will be dedicated to exploring the most widely used training techniques.

3.3 Convolutional networks

A convolutional neural network (CNN) is a particular type of multilayer feedforward network that consists of two different types of layers (convolution and pooling layers) connected alternatively. CNNs were initially inspired by the organization of the visual cortex of animals and are designed to process two- or three-dimensional data in a grid form, such as images and videos. For example, a colour image can be seen as a grid composed by pixel intensities in the three primary colour channels (red, blue and green) as inputs. The network breaks down the image in terms of simple properties (colour, textures, edges, contours, strokes, orientation) and learns them as representations in different layers.

The classical feedforward architectures have fully connected layers and they do not take into account the spatial structure of an image. CNNs were created to take advantage of the spatial structure of images by implementing three key ideas: local receptive fields, shared weights and pooling.

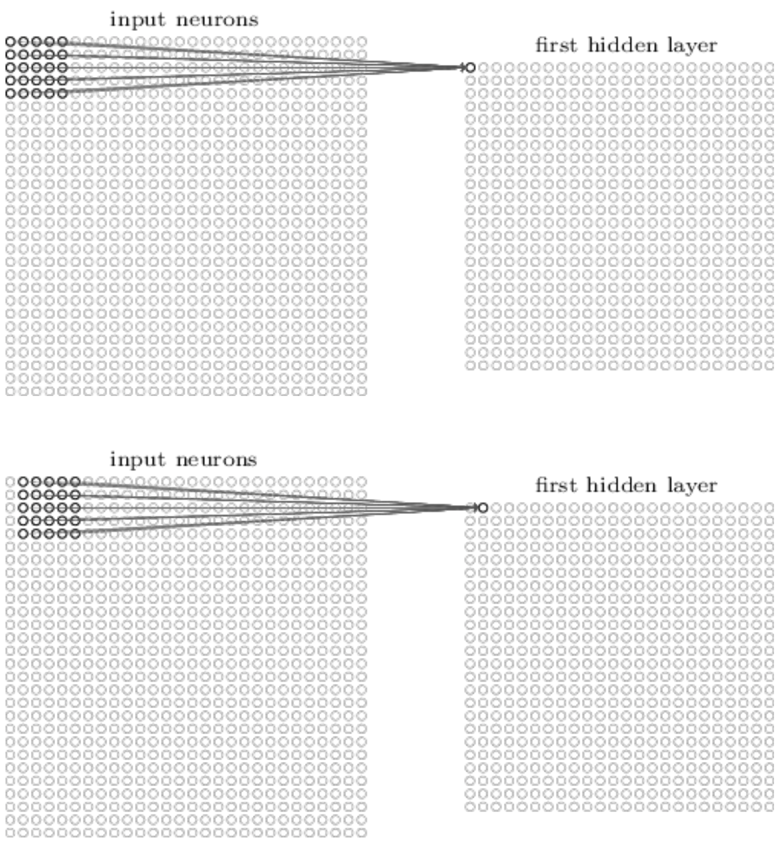
Local receptive fields

The input layer in a convolutional net consists of a matrix of neurons whose values correspond to the pixel intensities, rather than a single vector of neurons. The input pixels are connected to the first layer of hidden neurons, however each hidden neuron is connected only to a small and localised region of the input neurons, called local receptive field. Each hidden neuron learns to analyse its specific local receptive field and for each field there is a different neuron in the first hidden layer.

Figure 3.3 shows the relation between the two layers: the first neuron in the first hidden layer is connected to the receptive field (5x5 region) in the top-left corner of the input matrix (28x28 matrix) and the second hidden neuron is connected to the local receptive field that is obtained by moving the 5x5 square by one pixel to the right and so on for all the other neurons. The first hidden layer as a

result will be a 24x24 matrix. It is possible also to experiment with different stride lengths and move the local receptive field by two or more pixels to the right (or down).

Figure 3.3. Local receptive fields and related hidden neurons



Source: Nielsen (2015)

Local receptive fields are used because usually close values are highly correlated and form distinctive local motifs that can be easily detected.

Shared weights and thresholds

Considering the representation in figure 3.3, each hidden neuron has one threshold and 5x5 weights connected to its local receptive field. All the neurons in the first hidden layer have the same weights and bias and they detect exactly the same feature, but at different locations in the input image, which is essential for increasing both the invariance and the selectivity of the representation.

Each hidden layer can be called a feature map and the shared weights and bias related to a feature map are called kernel (or filter). For image recognition problems usually several feature maps and thus hidden layers are needed. A set of feature maps forms a convolutional layer and usually a network has more than one convolutional layer. The role of each convolutional layer is to detect local combinations of features from the previous layer. Notice that the hidden layers forming a convolutional layer are not connected one to the other, since they detect different features.

Pooling layers

In addition to convolutional layers, a CNN usually contains also pooling layers, which are responsible for merging semantically similar features into one, reducing the dimension of the representation and creating invariance to small shifts and distortions. A pooling layer is positioned immediately after each convolutional layer and it simplifies the information enclosed in the output coming from the convolutional layer by preparing a condensed set of feature maps. This helps to greatly reduce the number of parameters.

A common procedure for pooling is max-pooling, where the pooling unit outputs the maximum activation for each condensed receptive field in a feature map. Figure 3.4 shows the pooling procedure for the first feature map. Each max-pooling unit summarizes the output from the 24x24 first hidden layer by using a region of 2x2 neurons and taking the maximum activation in this region. The resulting layer will be a 12x12 matrix.

Figure 3.4. Pooling procedure

Source: Nielsen (2015)

Another popular pooling technique is L2 pooling, in which the information is condensed by taking the square root of the sum of the squares of activations in the 2x2 region instead of the maximum.

The simplest architecture will be similar to the one showed in Figure 3.5. The input layer encodes the pixel intensities of the image. This layer is followed by a convolutional layer composed of three feature maps, each responsible for detecting a specific feature across the entire image by sliding its local receptive fields and finding the shared weights and the bias. The result of this layer is passed through the pooling layer and condensed. A final fully connected layer connects every neuron from the pooled layer to each one of the output neurons.

Figure 3.5. CNN architecture with a convolutional and a pooling layer

28×28 $3 \times 24 \times 24$ $3 \times 12 \times 12$

Source: Nielsen (2015)

A typical CNN network can have many convolutional and pooling layers stacked one after the other. This will depend on the complexity of the image to be analysed and on the degree of abstraction needed. This type of network can be still trained with the classic backpropagation algorithm. The major advantages of a CNN are the minimal pre-processing required and the smaller number of parameters needed to get the same performance as a fully connected model. This results in faster training and it helps to mitigate the unstable gradient problem, to reduce overfitting and to produce more accurate results[8].

[8] For further details about CNNs, see Nielsen (2015) chapter 6 available at http://neuralnetworksanddeeplearning.com/chap6.html

The major applications for which CNNs have been used are speech recognition, document reading, handwriting recognition, object detection and face recognition in images and videos. They also brought a revolution in computer vision and became the dominant approach for recognition and detection tasks. In the recent years, they are being used in the development of new technology applications, for example in the vision system of self-driving cars and real-time vision applications in smartphones and cameras (LeCun et al., 2015).

Coding a CNN network in Matlab

CNN networks are powerful tools for image classification purposes. It is both possible to create a new network from scratch and train it on a large set of images or to use a pretrained network to learn new tasks from the already learnt features. The second approach is faster and easier to implement. The Deep Learning Toolbox in Matlab offers the possibility to deploy many pretrained networks with an intuitive interface and will be used for coding the examples in this chapter[9]. In this section, two simple examples about image classification with pretrained networks are presented.

Classifying an image using GoogLeNet

It is possible to classify an image into an object category using a pretrained CNN, such as GoogLeNet just in a few seconds. GoogLeNet is twenty-two layers deep network trained on ImageNet (a large visual database of over 14 million images) and is able to classify images into 1000 object categories. The code written in Listing 3.1 shows the steps needed to load and classify an image in .jpg format using GoogLeNet. The network takes the image as an input and will output a label for the object analyzed together with the probability of belonging to each object category.

Listing 3.1. Image classification using GoogLeNet

```
% load the pretrained model GoogLeNet network
net = googlenet;

% read the image to be classified (.jpg format)
```

[9] For further details about the Deep Learning Toolbox installation and usage in Matlab see https://www.mathworks.com/products/deep-learning.html

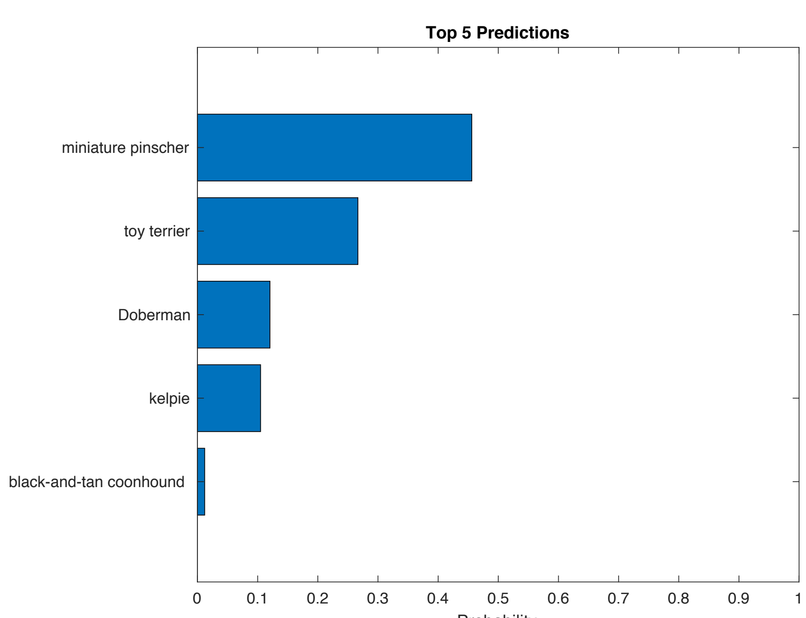
```
I = imread("dog2.jpg");

% Adjust the size of the image to 224x224x3 (input size of network)
sz = net.Layers(1).InputSize;
I = imresize(I,[sz(1) sz(2)]);

% Classify the image using GoogLeNet
[label,scores] = classify(net,I);
label

% Show the image and the predicted probability of the label
% classNames contains the names of the classes learned by GoogLeNet
classNames = net.Layers(end).ClassNames;
figure(1)
imshow(I)
title(string(label) + ", " + num2str(100*scores(classNames ==
label),3) + "%");

% Display Top 5 predicted labels and associated probabilities
[~,idx] = sort(scores,'descend');
idx = idx(5:-1:1);
classNamesTop = net.Layers(end).ClassNames(idx);
scoresTop = scores(idx);

figure(2)
barh(scoresTop)
xlim([0 1])
title('Top 5 Predictions')
xlabel('Probability')
yticklabels(classNamesTop)
```

Figure 3.6 shows the possible classification and related probability when the image of a dog is fed into the network. The network distinguishes also between different types of dog breeds.

Figure 3.6. Classification of a dog image

Since many categories considered by GoogLeNet are similar, it is useful to look at the top five predicted labels accuracy, as shown in Figure 3.7. The label prediction for similar objects is not always totally accurate as revealed by the histogram, but it can be still satisfactory, depending on the purpose of classification.

Figure 3.7. Histogram of the top-five predicted labels and their probabilities



Training an image category classifier using resnet50

A pretrained network can be used not only to classify a new image, but also as a feature extractor for training an image category classifier by leveraging the features extracted by the network. In this example the dataset considered is the Example Food Images dataset containing photographs of food belonging to nine different classes. The dataset can be found on the Mathworks website. The network used is the resnet50, a fifty layers deep network able to classify images into 1000 object categories.

The Listing 3.2 shows the steps necessary to train the image category classifier for the Example Food Images dataset. The dataset needs to be loaded and adjusted so that the number of images in each category is the same. The pretrained network is then loaded and the images are resized to have the same size required by the network. Now it is possible to extract the relevant training features from a deep

enough layer of the network and to train a multiclass SVM (support vector machine) classifier using those features. The test features are also extracted using the network and are then passed to the classifier to predict the labels for the test set. The accuracy of the trained classifier can be eventually measured by using a confusion matrix and displaying the mean accuracy. In the last step, one image from the test set is taken and the classifier is applied to it. The actual and the predicted labels for the image are then shown.

Listing 3.2. Image category classification using resnet50

```
%% unzip the dataset folder and load the image dataset
unzip('ExampleFoodImageDataset.zip')

imds = imageDatastore('ExampleFoodImageDataset', 'LabelSource',
'foldernames', 'IncludeSubfolders',true);

% summarize the number of images per category
tbl = countEachLabel(imds)

% Determine the smallest amount of images in a category
minSetCount = min(tbl{:,2});

% Limit the number of images to reduce the time it takes to run this
% example (limit to minSetCount)
maxNumImages = 100;
minSetCount = min(maxNumImages,minSetCount);

% Use splitEachLabel method to split the set into labels of the
% same size
imds = splitEachLabel(imds, minSetCount, 'randomize');

% Notice that each set now has exactly the same number of images
tbl_new = countEachLabel(imds)

%% load the pretrained ResNet-50 network

% Load pretrained network
net = resnet50();

% prepare training and test image sets
[trainingSet, testSet] = splitEachLabel(imds, 0.3, 'randomize');

%% pre-process the images for the CNN

% resize training and test sets to make them compatible with the
% input size required by the network
imageSize = net.Layers(1).InputSize;
augmentedTrainingSet = augmentedImageDatastore(imageSize,
trainingSet, 'ColorPreprocessing', 'gray2rgb');
augmentedTestSet = augmentedImageDatastore(imageSize, testSet,
'ColorPreprocessing', 'gray2rgb');

%% extract training features using the CNN
```

```
% extract features from a deeper layer using the activations method
% use the layer right before the classification layer
% (named 'fc1000')
featureLayer = 'fc1000';
trainingFeatures = activations(net, augmentedTrainingSet,
featureLayer, ...
    'MiniBatchSize', 4, 'OutputAs', 'columns');

%% Train A Multiclass SVM Classifier Using CNN Features

% Get training labels from the trainingSet
trainingLabels = trainingSet.Labels;

% Train multiclass SVM classifier using a fast linear solver and set
% 'ObservationsIn' to 'columns' to match the arrangement
% used for training features
classifier = fitcecoc(trainingFeatures, trainingLabels, ...
    'Learners', 'Linear', 'Coding', 'onevsall', 'ObservationsIn',
'columns');

%% Evaluate Classifier

% Extract image features from the test set using the CNN
testFeatures = activations(net, augmentedTestSet, featureLayer, ...
    'MiniBatchSize', 4, 'OutputAs', 'columns');

% Pass CNN image features to trained classifier
predictedLabels = predict(classifier, testFeatures, 'ObservationsIn',
'columns');

% Get the known labels
testLabels = testSet.Labels;

% Tabulate the results using a confusion matrix and display
% results in a chart
confMat = confusionmat(testLabels, predictedLabels);

% Convert confusion matrix into percentage form and show results
confMat = bsxfun(@rdivide,confMat,sum(confMat,2))

figure(1)
confusionchart(testLabels, predictedLabels);

% Display the mean accuracy
mean_accuracy = mean(diag(confMat))

%% Apply the Trained Classifier on one test image

testImage = readimage(testSet,1);
testLabel = testSet.Labels(1)

% Create augmentedImageDatastore to automatically resize the image
% when image features are extracted using activations
ds = augmentedImageDatastore(imageSize, testImage,
'ColorPreprocessing', 'gray2rgb');

% Extract image features using the CNN
imageFeatures = activations(net, ds, featureLayer, 'OutputAs',
'columns');
```

```
% Make a prediction using the classifier
predictedLabel = predict(classifier, imageFeatures, 'ObservationsIn',
'columns')
```

The results will be similar to the ones shown in Listing 3.3. Here `tbl` summarizes the number of images per category in the uploaded dataset, while `tbl_new` shows that after the adjustment the size is the same for all the labels. The next result is the confusion matrix (or error matrix), which shows in percentage terms the relationship between the predicted (row) and the actual (column) classes. The diagonal terms represent the correct classifications, while the off-diagonal terms show the wrong classifications. The mean accuracy is then computed using the diagonal terms. Finally, the test and the predicted labels are shown for the application on one test example.

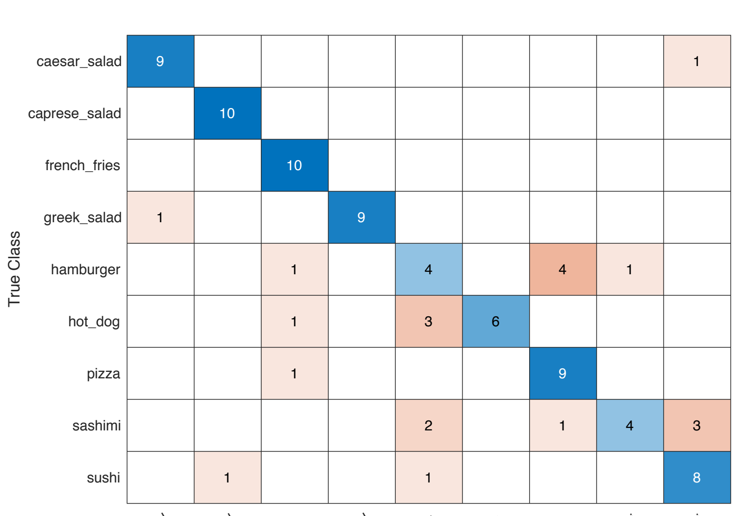
Listing 3.3. Results for the image category classification using resnet50

```
tbl =
  9×2 table
        Label        Count
    _____________    _____

    caesar_salad       26
    caprese_salad      15
    french_fries      181
    greek_salad        24
    hamburger         238
    hot_dog            31
    pizza             299
    sashimi            40
    sushi             124

tbl_new =
  9×2 table
        Label        Count
    _____________    _____

    caesar_salad      15
    caprese_salad     15
    french_fries      15
    greek_salad       15
```

```
    hamburger          15
    hot_dog            15
    pizza              15
    sashimi            15
    sushi              15

confMat =

    0.9000         0         0         0         0         0         0         0    0.1000
         0    1.0000         0         0         0         0         0         0         0
         0         0    1.0000         0         0         0         0         0         0
    0.1000         0         0    0.9000         0         0         0         0         0
         0         0    0.1000         0    0.4000         0    0.4000    0.1000         0
         0         0    0.1000         0    0.3000    0.6000         0         0         0
         0         0    0.1000         0         0         0    0.9000         0         0
         0         0         0         0    0.2000         0    0.1000    0.4000    0.3000
         0    0.1000         0         0    0.1000         0         0         0    0.8000

mean_accuracy =
    0.7667

testLabel =
  categorical
     caesar_salad

predictedLabel =
  categorical
     caesar_salad
```

It is also useful to depict the confusion matrix as a chart to better visualize the classification errors. This chart will be similar to the one shown in figure 3.8. The rows correspond to the predicted classes and the columns correspond to the true classes. The numerical terms simply represent how many images were classified as each specific class.

Figure 3.8. Histogram of the top-five predicted labels and their probabilities



3.4 LSTM networks

Recurrent neural networks are even harder to train than deep feedforward networks because gradients are not only propagated backwards through layers, but also through time, making the gradient extremely unstable and impeding learning. To solve this issue related to recurrent networks, it is possible to incorporate long short-term memory (LSTM) units into the RNN architecture and to augment the network with an explicit memory that remembers inputs for a long time. These units were first introduced by Hochreiter and Schmidhuber in 1997 to address the unstable gradient problem.

A typical LSTM architecture consists of a set of recurrently connected subnets, called memory blocks. Each block contains one or more self-connected memory cells and three multiplicative units (input, output and forget gates). The gates allow to store and access information over long periods of time.

To better understand the functioning of the network it is useful to look at the functioning of a LSTM memory block with a single cell, which is shown in figure 3.9. The three gates are nonlinear summation units that collect activations from both inside and outside the block and control the activation of the node by which they are multiplied (the multiplication is represented by a black circle). In detail, the input

and output gates are multiplied by the input node and the output of the cell, while the forget gate multiplies the previous state of the cell (or internal state). The activation of these gates is determined by a logistic sigmoid activation function (f). If the result of the activation is 1, the gate is opened and the flow from the multiplied node is passed through, otherwise if the value is 0, the gate stays closed and the flow is cut off. The input and output nodes are also passed through activation functions (g and h) in order to have the final output of each cell in the same dynamic range. Usually the activation functions chosen are tanh or logistic sigmoid. The weighted peephole connections from the internal state of the cell to the gates are represented with the dashed lines, while all the other connections have a fixed weight of 1, ensuring that the gradient can pass across many time steps without vanishing or exploding. The final output resulting from the memory block comes from the output gate multiplication.

Figure 3.9. LSTM memory block with a single cell

Source: Graves (2012)

In practice, new information enters the memory block from the input node and is run through the activation function g. The result is passed to the input gate, which decides if this information needs to be stored in the cell and passed on to the self-

connected memory cell. The memory cell acts like an accumulator, since it copies its own real-valued state and accumulates the external signal. This self-connection is multiplied by the forget gate, which controls the amount of information to be retained from previous time steps and when it is necessary to clear the content of the memory. The result is run through the activation function h and is passed forward to the output gate, which controls the final output.

The final LSTM network can be composed of many recurrently connected memory blocks. A simple example is shown in figure 3.10. This network consists of four input units, a hidden layer composed of two single-cell memory blocks and five output units. Notice that each block is fed with four inputs, but it produces only one output. The outputs from the two blocks are then passed to the final output layer, which produces the final result of the network. Only some of the connections resulting from the second input are shown in order to keep the representation simple and understandable. Looking at this picture, one can already grasp the complexity of the connections between the nodes even in a network composed by only two memory blocks.

Figure 3.10. LSTM network with two memory blocks

Source: Graves (2012)

Since the introduction of the original LSTM network, several variations have been proposed and it increased in popularity over the recent years. LSTM networks have shown a superior ability to learn long-range dependencies as compared to simple RNNs and they provide more accurate results. They have been widely applied to web document retrieval applications, translation of text into different languages, sentence generation to describe photographs (or image captioning) and handwriting recognition. LSTM networks can also answer questions that require complex inference. In one test example, the network is shown a 15-sentence version of the *The Lord of the Rings* and correctly answers questions such as “where is Frodo now?” (LeCun et al., 2015). For further details about LSTM networks see Graves (2012) and Lipton et al. (2015).

Coding a LSTM network in Matlab

The Deep Learning Toolbox in Matlab can be also used for forecasting a time series using a LSTM network. When forecasting future time steps of a sequence, it is possible to define the target outputs by shifting the training sequence by one time step to the right. The network predicts the time steps one at a time and updates the memory cell at each prediction.

In this example the time series of closing daily prices of the S&P 500 over four years is considered. The dataset is loaded and transformed into a row vector. It is then partitioned into the training and test subsets and it is standardized to prevent divergence. The target output vector for the training subset is specified by shifting the training sequence by one time step. Next the LSTM architecture is defined. There will be one unit in the input layer, 200 hidden units in the LSTM layer and one final output. The network is then trained over 200 epochs. The tuned network is used to forecast the future time steps using the test dataset. The network predicts the time steps one at a time and updates its state after each prediction.

Listing 3.4. Time Series Forecasting using a LSTM network

```
%% load sequence data
fileExtension = ".csv";
% Setup the Import Options
```

```
opts = delimitedTextImportOptions("NumVariables", 7);

% Specify range and delimiter
opts.DataLines = [2, Inf];
opts.Delimiter = ",";

% Specify column names and types
opts.VariableNames  =  ["Var1",  "Var2",  "Var3",  "Var4",  "Var5",
"AdjClose", "Var7"];
opts.SelectedVariableNames = "AdjClose";
opts.VariableTypes  =  ["string",  "string",  "string",  "string",
"string", "double", "string"];
opts  =  setvaropts(opts,  [1,  2,  3,  4,  5,  7],  "WhitespaceRule",
"preserve");
opts = setvaropts(opts, [1, 2, 3, 4, 5, 7], "EmptyFieldRule", "auto");
opts.ExtraColumnsRule = "ignore";
opts.EmptyLineRule = "read";


prices_data = readtable("^GSPC.csv", opts);
data = table2array(prices_data)'; % a row vector is needed
clear opts

% plot the data
figure(1)
plot(data)
xlabel("Day")
ylabel("Adjusted closing prices")
title("Daily adjusted closing prices of S&P500")

% partition training and test data
numTimeStepsTrain = floor(0.9*numel(data));

dataTrain = data(1:numTimeStepsTrain+1);
dataTest = data(numTimeStepsTrain+1:end);

% standardize data
mu = mean(dataTrain);
sig = std(dataTrain);

dataTrainStandardized = (dataTrain - mu) / sig;
```

```
% prepare predictors and responses
XTrain = dataTrainStandardized(1:end-1); % input values (xt)
YTrain = dataTrainStandardized(2:end); % desired output values (xt+1)

%% Define LSTM Network Architecture (layers)
numFeatures = 1;
numResponses = 1;
numHiddenUnits = 200;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits)
    fullyConnectedLayer(numResponses)
    regressionLayer];

% specify the training options (options)
options = trainingOptions('adam', ...
    'MaxEpochs',200, ...
    'GradientThreshold',1, ...
    'InitialLearnRate',0.005, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',125, ...
    'LearnRateDropFactor',0.2, ...
    'Verbose',0, ...
    'Plots','training-progress');

%% Train LSTM Network
net = trainNetwork(XTrain,YTrain,layers,options); % x = inputs, y =
outputs

% Forecast Future Time Steps (using a the standardized
% version of the test set)
dataTestStandardized = (dataTest - mu) / sig;
XTest = dataTestStandardized(1:end-1);
YTest = dataTest(2:end);

% Update Network State with Observed Values
% having the actual values of time steps between predictions,
% we can update the network state with the observed values

% initialize the network state by predicting on the training data
net = predictAndUpdateState(net,XTrain);
```

```
% predict on each time step
% (For each prediction, predict the next time step using the
% observed value of the previous time step)
YPred = [];
numTimeStepsTest = numel(XTest);
for i = 1:numTimeStepsTest
    [net,YPred(:,i)]=
predictAndUpdateState(net,XTest(:,i),'ExecutionEnvironment','cpu');
end

% Unstandardize the predictions using the parameters calculated earlier
YPred = sig*YPred + mu;

% Calculate the RMSE from the unstandardized predictions
rmse = sqrt(mean((YPred-YTest).^2))

% Plot the training time series with the forecasted value
figure(2)
plot(dataTrain(1:end-1))
hold on
idx = numTimeStepsTrain:(numTimeStepsTrain+numTimeStepsTest);
plot(idx,[data(numTimeStepsTrain) YPred],'.-')
hold off
xlabel("Day")
ylabel("Closing daily prices")
title("Forecast")
legend(["Observed" "Forecast"])

% Compare the forecasted values with the test data
figure(3)
subplot(2,1,1)
plot(YTest)
hold on
plot(YPred,'.-')
hold off
legend(["Observed" "Predicted"])
ylabel("Closing daily prices")
title("Forecast with Updates")

subplot(2,1,2)
stem(YPred - YTest)
```

```
xlabel("Day")
ylabel("Error")
title("RMSE = " + rmse)
```

Figure 3.11 shows the plot of the observed time series used for the training phase together with the forecasted values obtained with the test subset. Figure 3.12 looks in detail to the test subset and shows the plot comparing the forecasted and the target values and the plot showing the error at each time step (computed simply as the difference between predicted and target value) together with the RMSE (root mean square error). The RMSE assesses how well the network learns and it can be a useful metric when comparing the performance of different models.

Figure 3.11. Plot of observed and predicted time series

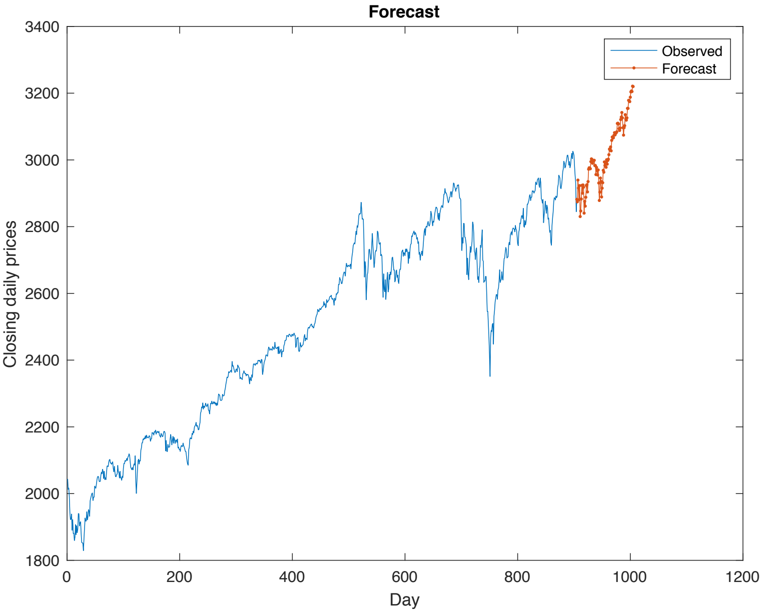
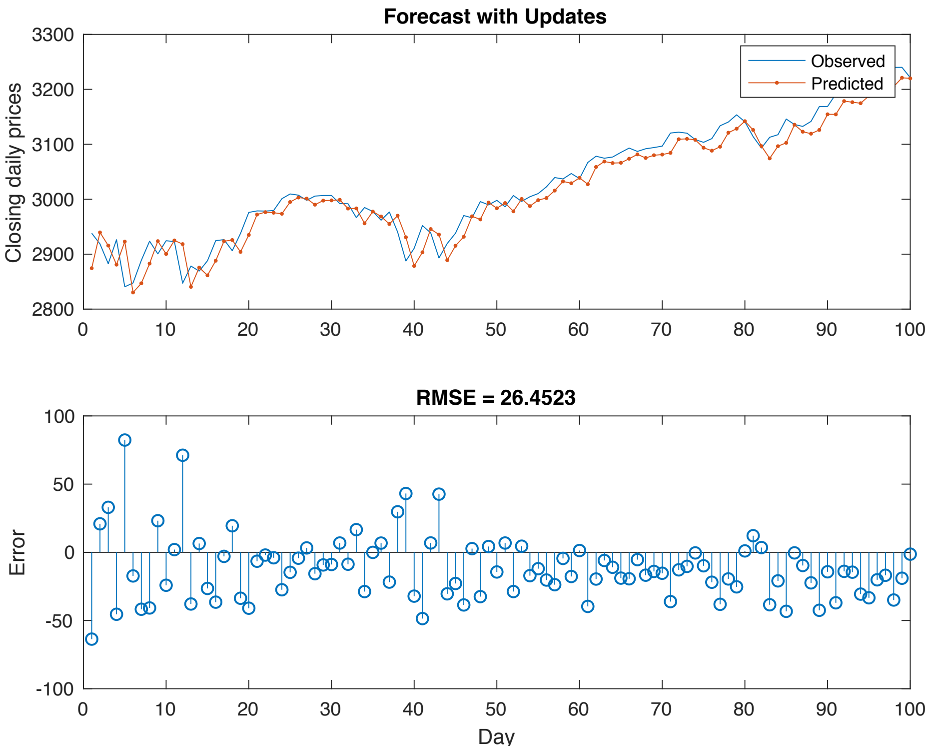


Figure 3.12. Plot comparing the forecasted and target values and resulting error

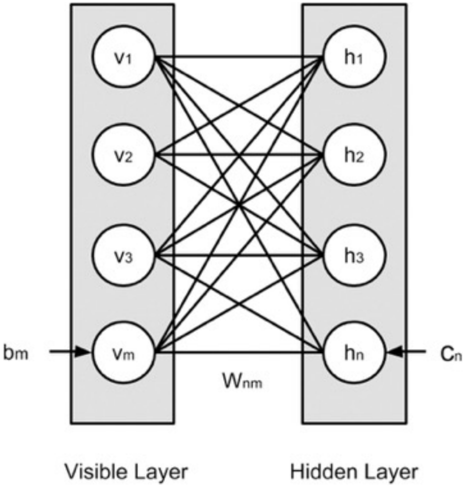


3.5 Other popular Deep learning models

<u>The Restricted Boltzman Machine</u>

The Restricted Boltzman Machine (RBM) is a type of stochastic neural network characterized by the ability to learn the probability distribution with respect to its inputs both in a supervised and unsupervised manner. As can be seen from Figure 3.13, the neurons are restricted to form a bipartite architecture, composed of two layers, a visible and a hidden layer. There is full and undirected connection between visible and hidden units, while there is no connection between units of the same layer. The network is trained to maximize the expected probability of the training samples. RBMs were at first proposed to simplify the topology of the network and increase its efficiency.

Figure 3.13. RBM with *m* visible and *n* hidden units

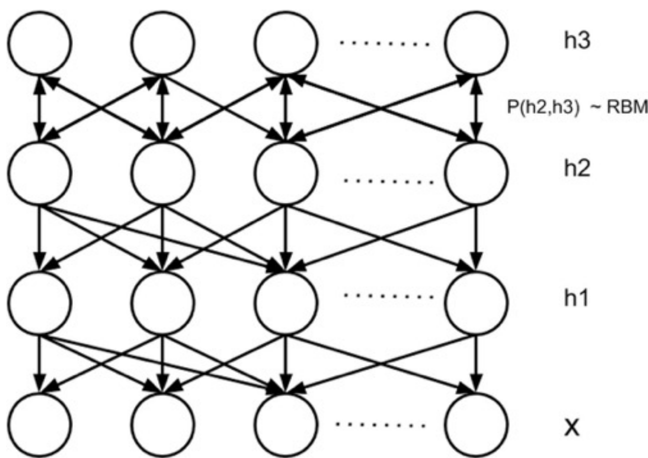


Source: Sengupta et al. (2020)

The Deep Belief Network

The Deep Belief Network (DBN) is a generative model composed of multiple layers of stochastic and latent variables, which represent the hidden features present in the input observations and are typically binary. DBNs are constructed by stacking a series of RBMs in order to further explore the dependencies between the hidden and visible variables.

Figure 3.14. DBN with *x* inputs and 3 hidden layers



Source: Sengupta et al. (2020)

As can be seen from figure 3.14, every two adjacent layers form a RBM, the visible layer is connected to the hidden layer of the previous RBM through directed connections in a top-down manner and the top two layers are non-directional. Being a generative model, the network learns the distribution of the data and is able to generate a sample based on its likeliness to happen. This means that the layers are trained sequentially: the lower RBMs are trained first, then the higher layers are trained. The lowest RBM learns the distribution of the input data, while the next RBM block learns the high order correlation between the hidden units of the previous hidden layer by sampling the hidden units. This process is repeated for each hidden layer until the top.

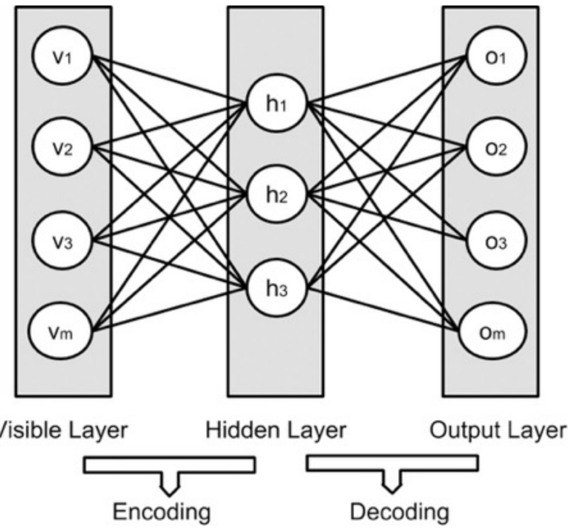
The training is done in two stages (a pretraining stage and a fine-tuning stage) by using the layer-wise-greedy-learning. In the pre-training stage, unsupervised training is carried out for feature extraction in the down-up direction. In this phase the initial weights are learned from the structure of the input data. The fine-tuning stage consists of a supervised learning in the up-down direction to further adjust the network parameters.

The pretraining phase provides a good set of initialised weights that are closer to the optimal weights compared to the randomly initialised weights and this improves the performance of the network, avoids overfitting and underfitting problems and effectively processes unlabelled data. DBNs have been used in many applications like phone recognition, computer vision, speech recognition and for pretraining DNNs and deep CNNs.

The autoencoder

The autoencoder is a three-layer neural network, that is used to efficiently code and reconstruct a dataset in order to reduce the data dimensionality. As can be seen from Figure 3.15, the input and output layer in this architecture contain the same number of neurons, while the hidden layer contains less units, since its purpose is to represent the inputs in a more compact form.

Figure 3.15. Autoencoder with 3 neurons in the hidden layer



Source: Sengupta et al. (2020)

The autoencoder is trained like a feedforward network by using the backpropagation algorithm and the training process is divided into two phases (encoding and decoding). In the encoding phase, the network tries to convert the inputs into an abstract (or hidden) representation with unsupervised learning and in the decoding phase, it reconstructs the same input from the hidden representation using supervised learning and it adjusts weights at each layer. The reconstruction accuracy is measured according to the minimization of the average mean square error of the reconstruction between the inputs and the reconstructed outputs. The main purpose of the autoencoder is to continuously extract useful features and filter useless information in order to enhance efficiency of the learning process. The encoded features of the autoencoder can effectively reflect the transformation invariant property. As an example, the autoencoder can be applied in image recognition to extract local features within a limited window of viewing in order to understand if a feature is present with certain probability and to pretrain CNNs. For further details see Liu et al. (2017) and Sengupta et al. (2020).

3.6 Techniques for improving training

The major problem for neural networks is overfitting and it is especially severe for deep networks having a large number of weights and biases. Techniques for detecting and reducing the effect of overfitting are needed. Several regularization techniques, such as L1 and L2 regularization, dropout, batch normalization, data augmentation and early stopping, have been developed for this purpose and are now widely used[10].

L1 and L2 regularization

The idea behind L1 and L2 regularization is to add an extra term, called regularization term, to the loss function in order to make the network prefer to learn small weights, all other things being equal. Large weights are allowed only if they considerably improve the first part of the loss function.

If the loss function is not regularized, the size of the weight vector is likely to grow and the length can become very large over time. The result is a higher probability of getting stuck and pointing in the same direction of the loss space after each weight adjustment. The direction can be wrong and stray away from the optimal minimum.

Another issue is that an unregularized network can use large weights to learn a complex model carrying a lot of information about the noise in the training data and it will respond with a large change in its behavior in response to small changes in the input. The risk for an unregularized network is to memorize the training data set without being able to generalize the information. On the other hand, a network with small weights is forced to ignore the effects of local noise in the dataset and to respond only to patterns often seen across the training set by building simple models that have a good generalization capability.

In the L2 regularization, the regularization term added to the loss function is the sum of the squares of all the weights in the network. The sum is scaled by a factor

[10] For further details about the regularization techniques, see Nielsen (2015) chapter 3 available at http://neuralnetworksanddeeplearning.com/chap3.html

$\frac{\lambda}{2n}$ where $\lambda > 0$ is the regularization parameter and n is the size of the training set. The L2 regularized loss function, can be written as:

$$E^{new} = E^{original} + \frac{\lambda}{2n}\sum_{w} w^2 \tag{3.3}$$

The regularization can also be seen as a compromise between finding small weights and minimizing the original loss function. The relative importance of the two factors depends on the value of λ, so that when λ is small the minimization is preferred and when λ is large small weights are preferred.

The bias terms are not included in the regularization formula, because having a large bias may be desirable since it allows flexibility in the network behavior.

L1 regularization requires the addiction of the sum of absolute values of the weights to the original loss function according to:

$$E^{new} = E^{original} + \frac{\lambda}{n}\sum_{w} |w| \tag{3.4}$$

This technique also penalizes large weights and makes the network prefer small weights. It tends to concentrate the weights in a relatively small number of high important connections, while the other weights are driven to 0.

Dropout

Dropout is a fundamentally different regularization technique. It does not rely on modifying the loss function and instead it modifies the network itself. The training process using dropout is done as follows. Part of the hidden neurons are randomly and temporarily deleted with probability p (usually $p = 0.5$ for hidden layers and $p = 0.8$ for the input layer) from the network as can be seen in Figure 3.16. The inputs are then forward propagated through the modified network and the result is backward propagated also through the modified network for a mini-batch of examples in order to update the weights and the thresholds. Subsequently, the process is repeated by first restoring the dropped neurons and then by choosing a new random subset of hidden neurons to delete with the same probability. The training process is done for a different mini-batch in order to update the weights and the thresholds. This process is repeated over and over so that the network will

learn a final set of weights and thresholds. During each iteration, the weights and thresholds are updated only for the neurons that are active during that iteration. This will ensure that there will be no co-adaptation (in which some neurons highly depend on other neurons) between the hidden neurons in one layer and will force each neuron to learn robust features alone.

When actually running the full network for the test phase, the activations should be scaled by multiplying them by the probability of keeping the neurons ($q = 1 - p$) so that the output at the test time matches to the expected output at the training time and the activations do not get too large. Having $q = 0.5$, twice the neurons in the test phase will be active and the activations for each neuron are cut in half to compensate for that.

The dropout technique can be implemented in a simpler and faster way by using inverted dropout. In this case the activations are scaled during the training phase, so that no changes in the network architecture are needed during the test phase. The activations are scaled by the inverse of the keep probability $1/q$ after training each mini-batch in order to prevent the activations from getting too large.

Figure 3.16. Dropout of half of the hidden neurons in a network

Source: Nielsen (2015)

The dropout procedure forces the network to learn more robust features and to be less affected by the loss of any individual connection in the network. Dropout

can also be viewed as training different neural networks and averaging their effects. The different networks will overfit in different ways and the net effect will be to reduce overfitting for the whole network.

To have even more accurate results, the dropout technique is utilized in combination with rectified linear units (ReLu), which can be applied to the output of a layer before or after dropout. A ReLu is a unit which employs a rectifier. The rectifier is a non-linear activation function and it computes $f(z) = \max(0, z)$. In practice it sets to 0 the neurons having a negative activation. Dahl et al. (2013) illustrate the benefits of using dropout and ReLu to train a deep neural network.

Batch normalization

Batch normalization is a popular technique that aims at improving the training of a deep neural network by stabilizing the distribution over a mini-batch of inputs to a given network layer during training. This is achieved by introducing additional network layers that control the mean and variance of each activation to be zero and one respectively. Then the normalized inputs are also scaled and shifted based on the trainable parameters.

The key motivation for the development of batch normalization was the reduction of the internal covariance shift (ICS). The ICS refers to the change in the distribution of activations in one layer caused by updates to the preceding layers. Such continual change negatively impacts training and leads to overfitting. Batch normalizations aims at reducing ICS.

New researches have found that batch normalization might not even reduce the ICS and instead it impacts the training process in a different fundamental way. This technique makes the error landscape significantly smoother and ensures that the gradients are more predictive and stable, allowing for faster convergence and for the use of much higher learning rates without the risk of divergence. For further details see Ioffe and Szegedy (2015) and Santurkar et al. (2018).

Data augmentation

Having a large training data set will considerably enhance the network performance. However, in practice there is a limit on the quantity of data that one can retrieve. A

possible solution is to artificially expand the training data and this approach is widely used especially in image and speech recognition applications for which is easy to modify the existing samples.

When analyzing images, the most popular augmentation techniques are flipping images vertically or horizontally, scaling or cropping by cutting out a region, rotating it by different angles, moving the object in different parts of the image, adding gaussian noise to distort the features and skewing the image.

When analyzing audios, it is possible to enrich the dataset by adding background noise, speeding up or slowing down the audio, changing pitch and shifting time by some seconds.

The general principle to follow is to expand the dataset by applying operations that reflect real-world variations and think about the human ability to recognize images and sounds even in presence of different distortions.

Early stopping

Overfitting can also arise from a too high number of training epochs. The most popular approach to prevent this issue is early stopping, which allows to identify the number of training epochs that can effectively improve the model performance. This strategy consists in computing the classification accuracy for the validation subset at the end of each epoch and keep track of how it changes. The classification accuracy simply measures the number of correct predictions over the number of total predictions.

Training is continued until there is enough confidence that the accuracy of the validation set has saturated. The saturation takes place when the accuracy is no longer improving. There is not a precise rule for choosing when to stop training, since training can improve by a little amount for many epochs. The identification of the epoch in which the accuracy saturates depends on the modeler judgement.

CHAPTER 4
Banking systems

The present chapter briefly introduces two important subjects related to banking systems. The first part of the chapter discusses the issue of systemic risk and its impact on financial stability. The second part illustrates stress-testing as a tool for assessing the financial stability of a banking system. This will help to clarify the key concepts which are essential for the comprehension and the application of deep learning to banking systems and in particular to stress tests that will be presented in the following chapters.

4.1 Systemic risk and financial stability

Systemic risk is defined by IMF, FSB and BIS (2009) as the risk of disruption to financial services that is caused by an impairment of all or parts of the financial system and has the potential to have serious negative consequences for the real economy. All types of financial intermediaries, markets and infrastructure can potentially be systemically important to some degree and negative externalities can emerge from a disruption or failure in one of them.

Modern banking systems and more generally financial systems are characterized by intricate linkages of claims and obligations between the balance sheets of banks and other financial institutions. The emergence of sophisticated financial products, such as credit default swaps (CDS) and collateralized debt obligations (CDOs), has increased the complexity of balance sheet connections even further (Gai and Kapadia, 2010).

The global financial crisis of 2007 and 2008 revealed the intertwined nature of the financial markets and the complexity of linkages. The initial troubles in the US subprime mortgage market rapidly escalated and spilled over to debt markets

across the world after the bankruptcy of Lehman Brothers. Due to fear and uncertainty, investors' risk appetite greatly diminished. Banks became less willing to lend money and started accumulating liquidity. As the short-term lending market run dry, interbank lending rates started to rise, causing a credit crunch (Allen and Babus, 2008).

This financial crisis can be explained by the presence of numerous interdependencies that facilitated amplified responses to shocks to the financial system and impaired the assessment of the potential contagion under distress. These interdependencies are present in the balance sheets of the different institutions and they can arise from both the asset and the liability side. As an example, asset linkages can result from holding similar portfolio exposures acquired in the interbank market and liability linkages result from sharing the same mass of depositors. When the system faces large shocks, denser interbank liabilities will facilitate contagion and propagate the shocks. For this reason, financial networks are robust yet fragile systems: the resilience of the system to the insolvency of any individual bank is enhanced because risks get reallocated and shared, however the connections create instability and systemic risk in case of large shocks (Acemoglu et al., 2015).

Another issue for which the financial system became vulnerable is the increased homogeneity due to similar diversification strategies. Risk became a commodity and it could be sold sliced and bundled. Securitization and derivatives were the most popular instruments to pass the risk between the different participants. Credit became structured and the length of credit chains increased dramatically. The interconnections multiplied and it became nearly impossible to track back the precise source and location of the underlying claims. Banks' balance sheets and their risk management strategies grew alike. All these diversification strategies by individual institutions generated a lack of diversity across the system as a whole. The complexity together with homogeneity lead to fragility and increased the probability of a collapse (see Gai et al. 2011 and Haldane 2013).

The structure of linkages between the different institutions can be captured by a network representation, which comprises a set of nodes and the links between them. The nodes represent the banks and their size, while the links are directed to

represent the mutual exposures of the assets and the liabilities. The network approach is important for assessing the financial stability and for capturing the potential externalities over the entire system coming from a single institution. It is essential to understand the network externalities in order to facilitate the macro-prudential financial supervision and to evaluate the resilience or fragility to contagion depending on the structure of the network (Allen and Babus, 2008).

Considering the network approach, it is possible to identify different sources of shock propagation in the banking system. In a simple model proposed by Haldane and May (2011), an external shock hits a single bank and wipes out part of its illiquid external assets (total loans made to ultimate investors representing the total size of the flow of funds from savers to investors through the banking system) and if this shock exceeds the capital reserve, the bank will fail. The shock will then propagate to the other connected nodes. A first source of propagation is given by liquidity shocks, arising from a generalized loss in the value of banks' external assets caused by a fall in market prices or a rise in expected defaults or fire sale actions at extremely low prices. Another mechanism of propagation comes from funding liquidity shocks. There is a diminished availability of interbank loans, as banks hoard liquidity for fear of lending to infected banks. The liquidity hoarding was a key feature of the global financial crisis.

Financial stability is defined by the World Bank as the resilience of financial systems to stress and economic shocks. The system efficiently allocates resources and manages the financial risks. When it is stable, it will absorb the shocks through self-corrective mechanisms, preventing disruptive effects on the economy[11]. There are several public policy interventions that can be made to reach the financial stability and the resilience of the system.

A key aspect of financial regulatory intervention resides in setting higher requirements for banks' capital and liquid assets in order to reduce idiosyncratic risks to the balance sheets of individual banks and to limit the potential spillovers in the system. The objective is strengthening the financial system as a whole. Setting higher capital buffers, strengthens the absorptive capacity of each bank to external

[11] See the complete explanation of World Bank at https://www.worldbank.org/en/publication/gfdr/gfdr-2016/background/financial-stability

shocks. Regulatory requirements on liquidity ratios aim at avoiding systemic liquidity spillovers arising from liquidity and funding liquidity shocks.

Another possible intervention relates to systemic regulatory requirements. Banks that are too big, connected or important to fail have higher buffers of capital and liquid assets and they contribute in greater proportion to the overall systemic risk. For this reason, they should have higher regulatory requirements.

It is also essential to regulate the derivatives market. The rapid growth in this market size and complexity contributed to the destabilization of the financial system during the crisis. The complex web of interactions can be diminished by centralizing the trading and clearing these instruments. Robust central counterparties interpose themselves between every bilateral transaction to reduce complexity and risk. The dimensionality of derivatives contracts can be reduced by eliminating redundant trades and by netting (offsetting multiple payment obligations into a single net payable or receivable).

The topology of the financial network has fundamental implications for the systemic risk and should be also taken into account. Regulatory incentives should promote diversity across the system in terms of aggregate balance sheets and risk management models. A modular structure of the system is also important to prevent contagion. According to this system configuration, too big banks should be split to limit their size or their activities (Haldane and May, 2011).

4.2 Stress-testing for financial stability

Central banks widely use stress tests for assessing financial stability. The output of stress tests carries essential information on the health and the vulnerabilities of the system or the entity considered. In addition, the reliability of the results influences the credibility of the authorities involved (Enoch et al., 2013).

Stress-testing is an important risk-management tool comprising various techniques for assessing resilience to extreme but plausible events. Traditionally it was employed to determine how a crisis scenario would affect the value of asset portfolios and later it has been applied to whole banks, banking systems and financial systems (Čihák, 2007).

Stress tests are used both by banks for internal risk-management and self-assessment purposes and by public authorities for assessing performance and risk profiles under adverse conditions and for determining appropriate regulatory requirements on capital and liquidity.

Stress tests can be divided into two main categories depending on the nature of the assessment and the consequences of the results. The first category refers to microprudential stress tests, which are forward-looking supervisory tools to assess the adequacy of individual banks' capital (or liquidity) depending on their portfolio risks. They are also important management instruments providing indications on the reliability of the internal systems designed for measuring their risks. The second category refers to macroprudential stress tests, which are focused on the overall financial stability rather than on a single financial institution. These tests aim at assessing financial vulnerabilities (such as high leverage, mispricing, concentration of risk and liquidity mismanagement) that can trigger systemic risks and compromise the financial stability of the whole economy (Adrian et al., 2020).

Considering the microprudential perspective, Schuermann (2014) argues that stress tests provide a useful indication on how much capital and liquidity is necessary for a bank to absorb losses and to support its risk-taking activities in case of a shock or to endure deteriorating economic conditions. Capital adequacy addresses the right side of the balance sheet (net worth) and liquidity refers to the left side (share of assets that can be readily converted into cash). According to the "Principles for sound stress testing practices and supervision" by Basel Committee (2009), stress tests are also important for the risk governance of a bank by:

- providing forward-looking assessments of risk;
- overcoming limitations of models and historical data;
- supporting internal and external communication;
- feeding into capital and liquidity planning procedures;
- informing the setting of a banks' risk tolerance;
- facilitating the development of risk mitigation or contingency plans across a range of stressed conditions.

Considering the macroprudential perspective, there are two main approaches that translate macroeconomic shocks and scenarios into financial sector variables. In the bottom-up approach, the authorities (central banks or supervisory authorities) define the macroeconomic shock to be considered and the banks are requested to evaluate its impact on their balance sheets. The results are then aggregated by the authorities in order to get the overall effect. The second approach is called top-down. Here the authorities design and calculate on their own the test for the banking system as a whole. According to the EBA (2018), bottom-up stress tests have the following characteristics:

- they are carried out by institutions using their own internally developed models;
- they are based on institutions' own assumptions or scenarios;
- they are based on the institution’s own data and high level of data granularity;
- they concern particular portfolios or the institution as a whole, producing detailed results on the potential impact of exposure concentrations, institution linkages and contagion probabilities to the institution’s loss rates.

On the other hand, top-down stress tests have the following characteristics:

- they are carried out by competent authorities or macroprudential authorities;
- they are based on general or systemic assumptions or scenarios designed by competent authorities and applicable to all relevant institutions. These authorities manage the process and calculate the results with less involvement of the institutions than in the case of the bottom-up stress test;
- they are based mostly on aggregate institution data and less detailed information, depending on the assumptions of the stress test;
- they enable a uniform and a common framework and comparative assessment of the impact of a given stress testing exercise across institutions.

After the financial crisis stress test became a crucial component of the supervisory and financial stability toolbox. The European Banking Authority (EBA) is responsible for ensuring the proper functioning of financial markets and it is mandated to monitor and asses market developments and potential vulnerabilities. The primary supervisory tool utilized is the biennial EU-wide stress test. It is initiated and coordinated by the EBA and it is an important input for the ECB's macroprudential policies. EU-wide stress tests are conducted in a bottom-up fashion and they assess individual banks' resilience to adverse events[12]. The European Central Bank (ECB) assures that the outcomes of bottom-up stress tests are credible by confronting banks with independent model-based estimates. It does so by taking a macroprudential perspective and performing top-down stress tests, which measure the system-wide impact of adverse shocks on the banking sector as a whole[13].

Stress tests are also run by international organizations to foster global growth. The International Monetary Fund (IMF) uses macroprudential stress tests in order to monitor and secure the global financial stability. The IMF assesses the resilience of financial systems and provides policy advices through the Financial Sector Assessment Program (FSAP) to its member states.

The supervisory authorities frequently use the international CAMEL rating system to determine the financial health of banks during an adverse shock. The factors affecting the performance are capital adequacy, asset quality, management capability, earnings, liquidity and sensitivity. As a guidance for an adequate capital and liquidity allocation by banks, the Basel Committee on Banking Supervision issued a set of regulatory standards and guidelines contained in the Basel I, II and III frameworks.

The Basel III framework is the most recent and it builds on Basel II by introducing important reforms to enhance risk coverage after the lessons learnt from the

[12] For further details about EU-wide stress tests done by the EBA, see https://eba.europa.eu/risk-analysis-and-data/eu-wide-stress-testing

[13] For more information on the macroprudential stress testing performed by the ECB, see https://www.ecb.europa.eu/press/key/date/2019/html/ecb.sp190904_2~4c8236275b.en.html

financial crisis (Basel Committee, 2011). It is essential that banks' risk exposures are backed by a high-quality capital base, hence the framework promotes the build-up of adequate buffers above the minimum that can be drawn in periods of stress. The main capital requirements measures proposed by Basel III are the common equity tier 1 (CET1) ratio and the capital adequacy ratio (CAR). The CET1 ratio can be defined as:

$$CET1\, ratio = \frac{CET1}{RWAs} \tag{4.1}$$

This ratio should be at least 4.5%. A great focus is put on CET1, because the common equity represents the highest quality components of a banks' capital.

The CAR can be expressed as:

$$CAR = \frac{Tier\ 1\ capital + Tier\ 2\ capital}{RWAs} \tag{4.2}$$

where $RWAs$ are the risk weighted assets (off-balance-sheet exposures weighted by risk). This ratio must be at least at a level of 8%. The numerator comprises both the Tier 1 and 2 capital to account for the coverage of potential losses and for protecting the depositors.

Strong buffers of capital are necessary to enhance the risk coverage but are not sufficient only by themselves. A bank needs a strong liquidity base as well. The framework establishes minimum liquidity requirements to achieve two complementary objectives. The first is to promote short-term resilience by ensuring that the bank has enough liquid assets to survive an acute shock of short duration. The liquidity coverage ratio (LCR) measures the amount of liquid assets needed and can be computed as:

$$LCR = \frac{high\ quality\ liquid\ assets}{total\ net\ cash\ flow} \tag{4.3}$$

The second objective is to promote long-term resilience by incentivizing a bank to fund its activities with stable funds. The Net Stable Funding Ratio (NSFR) measures the long-term resilience and it requires a minimum amount of stable sources of funding. It can be computed as:

$$NSFR = \frac{available\ amount\ of\ stable\ funding}{required\ amount\ of\ stable\ funding} \tag{4.4}$$

where the *available stable funding* is defined as the portion of capital and liabilities expected to be reliable over the time horizon considered by the NSFR, which extends to one year. The amount of the *required stable funding* of a specific institution is a function of the liquidity characteristics and residual maturities of the various assets held by that institution as well as those of its off-balance sheet (OBS) exposures (Basel Committee, 2014). Both LCR and NSFR ratios must be at least 100% according to Basel III.

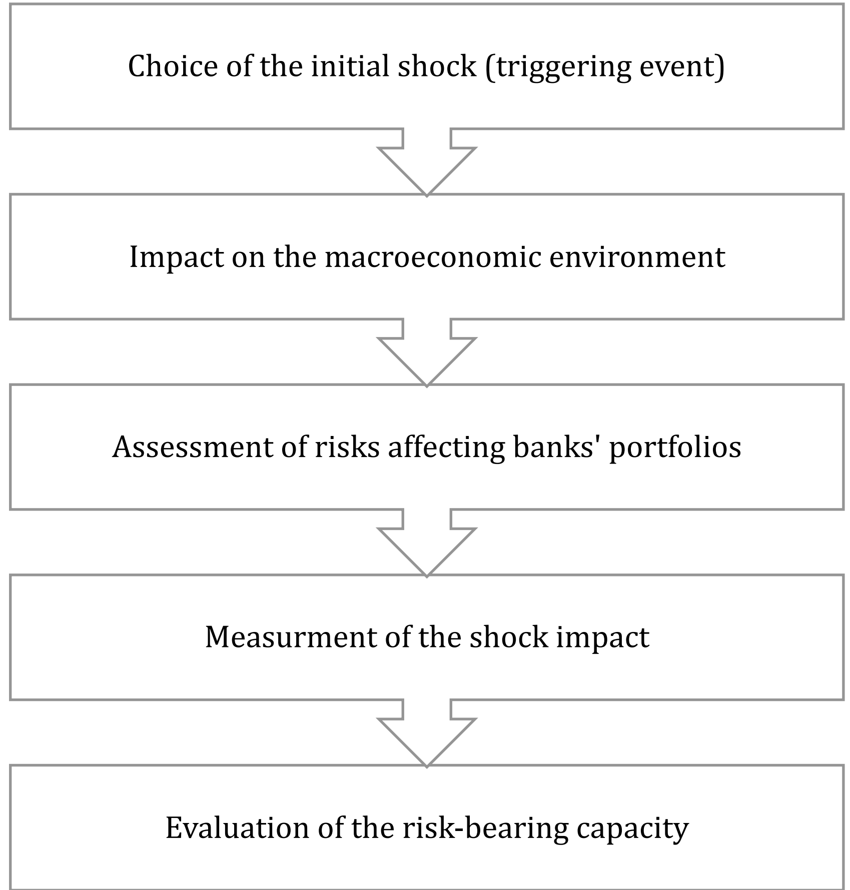
The framework introduces also leverage requirements in order to prevent excessive balance sheet leverage in the banking system that could amplify the downward pressure on asset prices during a crisis. The leverage ratio must be at least equal to 3% and it can be computed as:

$$liquidity\ ratio = \frac{Tier\ 1\ capital}{total\ exposures} \tag{4.5}$$

When a shock materializes, its impact is transmitted directly or indirectly to the financial sector. For banks, the main sources of risks can be broadly categorized as credit risk, market risk, liquidity risk and operational risk (ECB, 2006). Credit risk represents the largest source of risk for banks and it can be defined as the risk of a loss due to the borrower's failure to meet its obligations and repay a loan. Since, the lending activity is the core of the traditional banking business, credit risk has received closer attention in central bank stress tests. The EBA defines market risk as the risk of losses in on- and off-balance sheet positions arising from adverse movements in market prices. The main types of market risk are interest rate, equity, currency and commodity risk. The liquidity risk represents the chance for a bank to not be able to meet its short-term obligations. Operational risk is defined by Basel Committee (2004) as the risk of a loss resulting from inadequate or failed internal processes, people and systems or from external events. All those risks are considered in the Basel framework and are important for defining the minimum capital and liquidity requirements discussed above. Moreover, they are key components for the stress-testing procedure.

As a final consideration, it is useful to define a simple structure with the main steps that are usually followed when designing a stress test, as can be seen from Figure 4.1.

Figure 4.1. A simple structure of a stress test



The first step is to design a scenario and an initial shock or several shocks (e.g. a decline in GDP, a rise in interest rates, a crash in equity markets or a spike in oil prices). The initial shocks are collected in a scenario, which can be historical, hypothetical, probabilistic or reverse-engineered. Historical scenarios replicate significant market events that happened in the past (such as the 1987 stock crash, the 1998 emerging markets crisis or the 2000 tech bubble burst). Hypothetical scenarios do not match historical events and aim to capture plausible events that have not occurred yet. Probabilistic scenarios are constructed on the basis of the empirical distribution of the relevant risk variable, corresponding to extreme

percentiles in the distribution. Reverse-engineered scenarios are constructed to match a predefined amount of losses to be endured by the financial sector. The key point for having meaningful stress scenarios is to incorporate plausible low-probability shocks, which represent extreme realizations of the underlying risk factor.

The second step is to describe the impact of the initial shock on the macroeconomic environment. A macroeconomic model that links external shocks to key macroeconomic variables, such as GDP, interest rates, exchange rate, and other variables is chosen. This will provide an internally consistent representation of the full economy under stress.

The third step consists in addressing the different types of risks that affect a bank's portfolio, such as credit risk, market risk, liquidity risk and operational risk. The choice can fall on a single risk or a combination of two or more risks.

The fourth step requires to measure the impact of the shock on the bank's balance sheet. This can be done by linking the macroeconomic variables to banks' asset quality using a satellite model.

In the fifth step, the output of the stress test can be combined with other pieces of information to assess the strength of the bank or the banking sector. The impact on the single banks can be expressed in terms of a variable such as capital adequacy or capital injection as a percent of GDP. For further details see ECB (2006) and Čihák (2007).

CHAPTER 5
Deep Learning applications for banking systems

The present chapter discusses how the use of artificial neural networks can improve several real-world financial applications, which often have a non-linear behaviour. Banks and regulatory authorities can benefit from the implementation of both simple artificial neural networks and more complex deep neural networks in several areas, such as credit scoring, bankruptcy prediction, financial crisis prediction and stress-testing. The importance of these applications for the banking system is addressed and some outstanding researches are briefly presented.

5.1 Financial applications of ANNs and DNNs

Many real-world financial applications have a nonlinear and uncertain financial behaviour which changes over time. These problems stimulated a growing interest in machine learning techniques and especially in artificial neural networks. Several researches show that the accuracy of these methods in many cases is superior to traditional statistical methods (parametric and nonparametric) in dealing with financial problems, especially regarding nonlinear patterns. What makes multilayer ANNs truly appealing is their ability to capture complex and nonlinear interactions among the variables. The more hidden layers are in a neural network, the more complex is the interaction effect that can be modeled (Bahrammirzaee, 2010).

Financial prediction problems are of keen interest both for practical and theoretical purposes. Financial prediction problems, such as design and pricing of securities and construction of portfolios and risk management, often involve large data sets with complex data interactions that are difficult or impossible to specify in a full economic model. Deep learning methods can detect and exploit complex non-

linear interactions in the data that are invisible to any existing financial economic theory and could produce more accurate predictive outputs than standard methods (Heaton et al., 2016).

The following sections will present some of the most important areas of application of ANNs and deep learning to banking systems: credit scoring, bankruptcy prediction, financial crisis prediction and stress-testing.

5.2 Credit risk evaluation and credit scoring

The assessment of credit risk became the major focus of the banking industry after the global financial crisis and the stricter capital requirements of the Basel III framework. Credit risk is still the largest risk, difficult to manage and to evaluate. The ability to discriminate good customers from bad ones is crucial for all credit-granting institutions, such as commercial banks and certain retailers. The need for reliable models that predict accurately the possibility of default is crucial so that the interested parties can take either preventive or corrective action. Consequently, an accurate prediction of credit risk is very important for sustainability, profit of enterprises and a more efficient use of economic capital in business. The major advantage for the bank relates to sounder lending decisions, which in turn can result in significant savings (Yu et al., 2008).

An improvement in default prediction accuracy of even a small amount can lead to large savings. In addition to avoiding potentially troubled borrowers, the credit risk evaluation can help in estimating a fair value of the interest rate of a loan that reflects the creditworthiness of the counterparty and in carefully assessing the credit risk of the loans portfolio. The credit risk assessment in practice requires the computation of a loss level for which there is a probability of 1% that the loss incurred in the portfolio will exceed that level in a particular time period (Atiya 2001).

Financial institutions and their portfolios of loans increased considerably in scale and complexity over the past years. At the same time, advances in information technology have lowered significantly the costs of acquiring, managing and analyzing large amounts of data and it became possible to build more robust and

efficient techniques for credit risk management (Pacelli and Azzollini, 2011). Credit scoring models have been developed for the credit granting decision with the aim of assigning credit applicants to one of two groups: a good credit group that is likely to repay the financial obligation, or a bad credit group that should be denied credit because of a high likelihood of defaulting on the financial obligation (West, 2000).

Models based on credit scoring can be used to predict the potential risk of a credit portfolio. Banks evaluate credit both for corporations and individual consumers. Credit scoring is divided into two distinct types. The first is application scoring, for which the task is to classify credit applicants into good or bad risk groups depending on their probability of default. The data used for modeling generally consists of financial information and demographic information about the loan applicant. The second type is behavioral scoring and it is used for existing customers. Payment history information is also used here along with other information by taking into account the customer's payment pattern on the loan (Khashman, 2010).

The traditional approach is based on statistical models. The most popular ones are the linear discriminant analysis (LDA) technique and the logistic regression (LR) approach. They use financial statement data and financial ratios.

The principal concern of credit scoring applications is the necessity to increase the scoring accuracy of the credit decision. An improvement in accuracy of even a small amount can translate into considerable future savings. For this reason, novel models have been explored. ANNs were found to be efficient methods for credit scoring and they have outperformed in many cases the traditional methods. Their strength resides in their non-linear nature, which is particularly useful in modelling complex non-linear relationships between the dependent and independent variables. The multi-layer perceptron is the most frequently used architecture.

Several researches have tested the employment of ANN models. West (2000) explores the credit scoring accuracy of five different network models and benchmarks the results against LDA and other traditional methods. It compares the performance of the multilayer perceptron to other networks like the radial basis function and fuzzy adaptive resonance in order to explore if there are more efficient networks than the MLP. The author uses two real-world datasets containing

examples of loan applications of individuals, the German credit dataset and the Australian credit dataset. For each applicant, a set of variables describes credit history, account balances, loan purpose, loan amount, employment status, personal information, age, housing and job. The paper employs a data mining strategy, using all the variables as raw data. The final output given by the network is a number representing the credit score. The results suggest that all the neural network models considered achieved improvements in accuracy with respect to traditional ones. The classic MLP, however, is not the most accurate and it is suggested to try different architectures and topologies depending on the available data.

Another more recent research that explores credit scoring models for individual loans is the work in Khashman (2010). The dataset employed is again the German credit dataset. The paper describes a credit risk evaluation system that uses a multilayer perceptron with one hidden layer based on the back-propagation learning algorithm. The paper investigates the impact of change in size of the hidden layer, different learning rates and the training-to-test ratio on the final result.

Several other studies concentrate on corporate credit scoring. Atiya (2001) develops a bankruptcy prediction model using the classic MLP network. The author tackles the problem of the inputs to be used and he introduces a novel set of input indicators extracted from the stock price of the firm in addition to the financial ratios. The reason is that a problem faced by a firm will typically be reflected in the stock price well before it shows up in its balance sheet and income statement. The stock-price-based indicators considered are the volatility, the change in price and the price-cashflow ratio. The comparative advantage of the financial ratio and equity-based system over the financial ratio system is tested on historical data from solvent and defaulted US firms and it is found that the addition of the new indicators improves the prediction considerably.

Angelini et al. (2008) describe an application of neural networks to the credit risk assessment of Italian small businesses. They develop two neural network systems, one with a standard feed forward network, while the other with a special purpose architecture. The inputs used are financial ratios drawn from the balance sheet of the firm, ratios calculated by analyzing the credit positions with the supplying bank and ratios representing the overall Italian Banking System. The output y of the network, which is a real value in the range [0,1], is interpreted as

bonis if $y < 0.5$ and, otherwise, the example is classified as default. The application is tested on real-world data and it is shown that both neural networks can be very successful in learning and estimating the default tendency of a borrower, provided that careful data analysis, data pre-processing and training are performed.

There are several problems with financial data of this kind that have to be considered, as real-world data is often noisy and incomplete. Angelini et al. (2008) also investigate in detail the problems that one could encounter and what pre-processing operations should be carried out. When there are missing or wrong values it is useful to replace them with other values, such as the arithmetical mean of the field or the upper limit of the normalization interval. Fields with too many missing or wrong values should be erased from the dataset. The data set should be normalized with an appropriate technique. The authors suggest using the logarithmic formula, since it is more flexible than a simple linear transformation. A correlation analysis between the different ratios should also be performed and the strongly correlated variables should be removed.

In relation to the normalization issue, Khashman (2010) explains that the numerical input values representing the attributes of a credit applicant vary marginally in value, and if a simple normalization process is applied to the whole dataset, as an example by dividing each value in the set by the largest recorded value, then much information would be lost across the different attributes. Therefore, the normalization of credit application input data should be carefully performed, while maintaining the meaning of each attribute.

5.3 Bank insolvency and bankruptcy prediction

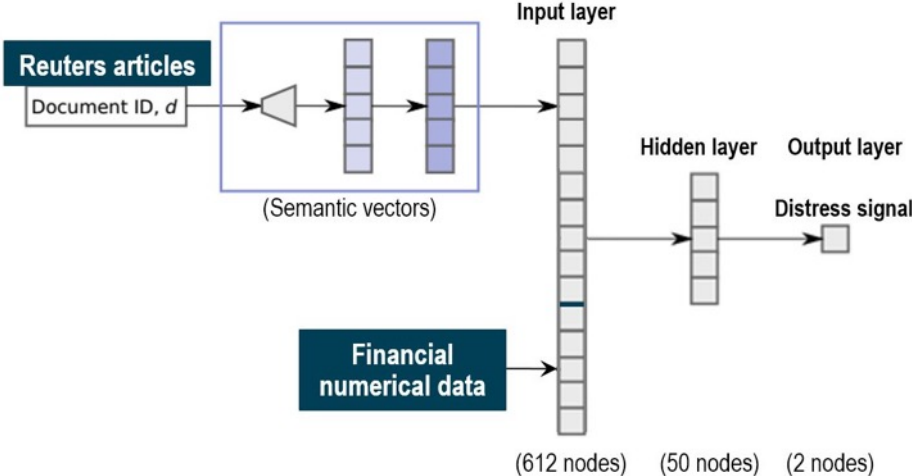
The prediction of bankruptcy for financial firms and especially for banks has been widely investigated since the late 1960s. Stockholders, senior management, creditors and auditors are all interested in predicting bankruptcy because it affects similarly all of them. The health of a bank in a highly competitive business environment relies on how financially solvent it is at the beginning; its ability, flexibility and efficiency in creating cash from its operations; its access to capital markets; and its financial capacity and resilience when faced with unplanned short

falls in cash. As a bank or firm becomes more and more insolvent, it gradually enters a danger zone and changes must be made to its operations and capital structure in order to keep it solvent (Kumar and Ravi, 2007).

Artificial neural networks have been successfully employed as well for predicting bankruptcies. Al-Shayea et al. (2010) aim to predict bank insolvency before the occurrence of bankruptcy using neural networks in order to enable all parties to take promptly corrective action. The model works as an early warning system and monitors solvency. A dataset of financial ratios of Spanish banks is inputted into a two-layer feed-forward network with sigmoid hidden neurons and linear output neurons. The financial ratios chosen measure liquidity, self-financing capability, ability to make profits and to produce positive cash flows. The results show that even a simple model with a limited amount of data like this has a good ability to learn the patterns corresponding to the financial distress of a bank.

The more recent work of Cerchiello et al. (2017) attempts to incorporate textual information of financial news in addition to numerical data with the purpose of enhancing the performance of classifiers of bank distress. Understanding bank distress is an area of research where textual data holds promising potential due to the frequency and information richness of financial news. Indeed, central banks are starting to recognize the utility of textual data in financial risk analytics. The authors construct a dataset containing both numerical (information on bank-level balance sheet and income statement data, as well as country-level banking sector and macro-financial data) and textual (news articles from Reuters online archive) information by matching the news dates to the numerical data. As can be seen from figure 5.1, the framework processes textual data through an unsupervised neural network model converting the documents sentences into sentence vectors. The retrieved sentence vectors are then joined with the financial numerical data in a unique input vector and fed to a three-layer fully connected feedforward neural network. The network produces a distress signal as an output. The experimental results confirm that the integration of numerical and textual data amplifies the prediction capability of the model compared with textual data alone.

Figure 5.1. Structure of the framework



Source: Cerchiello et al. (2017)

The use of textual disclosures in ANN models has raised interest also in predicting corporate bankruptcies. Mai et al. (2019) introduce a deep learning model for corporate bankruptcy forecast using textual disclosures, a form of unstructured and qualitative data. Textual data plays an important role in how information is conveyed to the public. For example, a vast proportion of public firm's annual filings to regulatory agencies are textual disclosures. Also, policymakers and market participants consume a large amount of financial reports and news articles every day. As a large amount of unstructured data is injected into the market every day, investors, regulators and researchers demand more intelligent models to digest such information.

The authors construct a comprehensive bankruptcy database of 11,827 U.S. public companies. The database consists of numeric variables generated from accounting and stock market data that may reflect the company's liability, liquidity and profitability status. In addition, textual data is taken from the Management Discussion and Analysis (MD&A) section of the 10-K SEC filing. The SEC mandates public companies to include an MD&A section in their annual reports. This section contains a narrative explanation of the firm's operations in a way that an average investor can understand. It also serves as a qualitative disclosure for investors to make more accurate projections of future financial and operating results. A natural language processing model is used to pre-process, simplify and extract relevant

features from the textual information. The deep learning system designed for the prediction task is a feedforward model that maps the inputs (numeric and text features) to a binary output (bankruptcy or not). The model is trained with the stochastic gradient descent. The final result shows that deep learning models yield superior prediction performance in forecasting bankruptcy using textual disclosures in conjunction with traditional accounting-based ratio and market-based variables.

5.4 Financial crisis prediction and early warning systems

A global financial crisis can arise from a series of local market shocks and subsequently evolve into a worldwide economic crisis due to the interconnections of the financial markets. In other instances, a crisis can start from a single economy whose size is large enough to generate instability also in the other countries. An example is the subprime crisis that started in the United States and evolved into a sovereign debt crisis in several European countries (Chatzis et al., 2018).

There are several shapes and forms that a financial crisis can take, such as currency crises, sudden stops, debt crises and banking crises. A currency crisis involves a speculative attack on the currency that can result in a devaluation or force the authorities to defend their currency by expending large amount of international reserves or sharply raising interest rates. A sudden stop can result from a large and unexpected fall in international capital inflows or a sharp reversal in aggregate capital flows to a country. A debt crisis is associated with adverse debt dynamics for which a country is not able to honor its domestic or foreign obligations. A banking crisis involves bank runs and failures that can induce the banks in a system to suspend the convertibility of their liabilities or to ask for the intervention of government to extend liquidity and capital assistance on a large scale (IMF, 2013).

Early Warning Systems (EWSs) are models that produce clear and timely signals of the occurrence of an economic crisis. They are valuable tools for policymakers in their effort to contain the contagion risk and to anticipate a global economic crisis. Hence, EWSs can help the policy makers in uncovering the possible

vulnerabilities of the economy and taking preemptive actions to diminish those risks (Chatzis et al., 2018).

Artificial neural networks have been recently applied to solve also this type of prediction problem. Fioramanti (2008) develops an early warning system to predict a sovereign debt crisis in a set of developing countries. He uses a large panel dataset for 46 emerging countries of internal (GDP growth, inflation, interest rate), external (US treasury bill interest rate, overvaluation, exchange rate agreement, degree of openness) and debt-related explanatory variables (average maturity, total external debt, short-term external debt, interest on external debt). A country is defined to be in a debt crisis if it is classified as being in default by Standard & Poor's or if it receives a large non-concessional IMF loan. This definition is used to specify the crisis episodes used to define the dependent variable. The data is fed into a multilayer perceptron having a hidden layer of three neurons. The output produced is a binary number predicting if there will be a debt crisis or not. The final result shows that the MLP can improve the performance of EWSs and it could be used together with traditional methods to make the final judgment.

Aydin and Cavdar (2015) develop an early warning system to predict a financial crisis in Turkey. They use seven key monthly macroeconomic and financial indicators of the Turkish economy as inputs (US Dollar, gold prices, Istanbul 100 Index, wholesale price index, money supply, domestic debt stock and composite leading economic indicators index). The data is fed into a feedforward ANN with two hidden layers. The final output consists of the prediction of the seven indicators after the next 24 months. The results show that this simple model has a good ability to learn the patterns corresponding to a financial crisis.

Chatzis et al. (2018) investigate the use of different machine learning techniques for forecasting a stock market crisis. They construct a database containing various financial indicators (such as the yield of the 10-year government bond, the exchange rate against the US dollar, oil price, gold price and VIX index) focusing on liquid markets, where the transmission of extreme events is better depicted in the pricing patterns. Several countries are included to cover the three most important financial markets, namely America, Asia and Europe. The authors also identify all crisis events for each country. A significant global crisis event

happens if the return of the Stock Index is below the first percentile of the associated empirical distribution of returns. They created two binary dependent variables, one pertaining to a one-day predictive horizon and one pertaining to a 20-day predictive horizon. A series of models including classification trees, support vector machines, random forests, neural networks, extreme gradient boosting and deep neural networks are applied to the dataset and their performance is compared. Deep neural networks with five hidden layers consistently outperform the rest of the employed approaches and can be a good starting point for developing an early warning system.

5.5 Dynamic balance sheet stress-testing

Financial stability is a central aspect for the economic prosperity of countries and individuals. Regulatory authorities and international organizations performed stress testing exercises to assess the resilience of the banking system long before the financial crisis of 2007 but failed to predict the unprecedented economic turmoil after Lehman's default. After that, more rigorous stress testing exercises have been developed. Yet, the current stress testing frameworks are usually composed of a feed forward shock engine, which is not able to capture the relationships nexus of the highly interconnected financial system and the associated feedback loops in the macro environment.

The EBA EU's wide stress-testing is a bottom up exercise, which covers only specific risks on individual banks' balance sheets based on a macro scenario with simplified assumptions. One of the major weaknesses in EBA methodology is the static balance sheet assumption, which requires assets and liabilities to remain constant over the horizon considered without acknowledging for management actions or new loans. Macroeconomic feedback effects, such as the impact of a significant institution becoming insolvent in the macro economy, usually are not considered in these frameworks. Stress tests under this structure serve to challenge the recovery plans of banks and to assess their viability. However, they are inadequate to provide early warning signals. Another issue of stress tests is their reliance on regulatory ratios like capital adequacy ratio, which in turn is highly dependable on the estimation of risk weighted assets (RWAs). Relying on the risk

weights applied internally by the financial institutions under the Basel Framework can lead to an underestimation of the capital needs, since RWAs are not able capture the hidden risk in the complex portfolio structure of a bank and there is significant variability in the type of internal models used (Petropoulos et al., 2019).

A significant amount of granular information on banks has been collected after the crisis by regulators, however they have yet not explored machine learning techniques in order to extract more information regarding the risks in their banking systems. Petropoulos et al. (2019) give the first contribution to this field with the application of deep learning to a dynamic balance sheet stress-testing framework. Financial or macroeconomic shocks are propagated to banks' balance sheets by simultaneously training a deep neural network with macro and financial variables. The model is capable of capturing more information hidden in a big dataset and it accounts for complex non-linear relationships that materialize under adverse macroeconomic conditions and financial distress. This framework independently assesses the banking system without relying on the single banks' estimations. Only publicly available data is used and the model is developed in a uniform way making the process of validation and error correction more feasible to be performed centrally by regulators.

The authors compare the performance of the deep learning approach to other well-known stress testing frameworks, such as the constant balance sheet approach and the dynamic balance sheet approach with satellite modelling. The prediction error of the Capital Adequacy Ratio (CAR) drops significantly under the deep learning approach, due to its better performance in simulating the one year ahead profit and loss evolution of the financial institutions. For this reason, the deep learning framework can become a powerful tool for macro prudential stress-testing and can improve the signaling power of an early warning system in order to predict future financial crises and individual bank's failures.

CHAPTER 6
Application to dynamic balance sheet stress-testing

The present chapter deals with a final case study analysis, in which the application of a neural network to dynamic balance sheet stress-testing is performed by using real US data. The first section illustrates how the data was collected and pre-processed, while the second section explains how the model was designed, giving an overview about the choice of a suitable architecture and hyperparameters in order to enhance the model's generalization and predictive capabilities.

6.1 Data collection and pre-processing

This case study analysis was inspired by the work of Petropoulos et al. (2019), which was presented in the previous chapter. Instead of predicting nine output variables that capture the profitability structure of a bank, this analysis focuses on a simpler problem and tries to predict only the one-year ahead CAR ratio, which is a key ratio that regulators monitor closely when determining the financial health and solvency of a bank.

In order to solve the prediction problem of the CAR ratio, quarterly information of various financial and macroeconomic indicators was collected from different sources and databases. The relevant stress financial variables for simulating the profitability and the risk weighted assets of each active financial institution were collected from the database of the Federal Deposit Insurance Corporation (FDIC), an independent agency created by the US Congress in order to maintain the stability and the public confidence in the financial system (Petropoulos et al., 2019). The relevant macro-economic variables were collected from Federal Reserve Economic Data (FRED) database, supported by the Research division of

the Federal Reserve Bank of St. Louis. Macro-economic variables have to be considered in the model, since shocks can propagate from the macro economy into the financial institutions' business models through non-linear channels. The dataset covers a nine years' period from 2010 to 2018 with quarterly information for a total of more than 27000 records. In order to capture the unique characteristics of each financial entity, 3-year lags are also included for each variable, for a total of 80 input variables.

Two of the model variables have to be computed by hand using other variables found in the database. The risk weight density is computed according to the definition of Basel Committee (2016) as:

$$risk\ weight\ density = \frac{leverage\ ratio}{Tier\ 1\ Capital} * risk\ weighted\ assets \tag{6.1}$$

The S&P 500 quarterly returns are computed from the monthly prices of S&P 500 Total Return Index according to:

$$r_t = \ln\left(\frac{P_t}{P_{t-3}}\right) \tag{6.2}$$

The financial and macro-economic variables are summarized in Table 6.1.

Table 6.1. Description of the model variables

Variable	Description	Type
net_loan	Net loans and leases exposure	Financial
dep	Total deposits	Financial
loss_allow	Loss allowance to loans	Financial
yield_ea	Yield on earning assets	Financial
fundc_ea	Cost of funding earning assets	Financial
inc_aa	Noninterest income to average assets	Financial
CAR	Total risk-based capital ratio	Financial
tot_asst	Average total assets	Financial
tot_eq	Average total equity	Financial
tot_loan	Average total loans	Financial
risk_dens	Risk weight density	Financial
GDP_growth	Gross Domestic Product growth	Macro-economic
export_growth	US real exports of goods and services growth	Macro-economic
debt_GDP	US public debt to GDP	Macro-economic
govex_GDP	US government expenditure to GDP	Macro-economic
inflat	Implicit price deflator as a measure of US inflation	Macro-economic
HPI_growth	House Price Index growth	Macro-economic
unemp	Unemployment rate (age 15-64)	Macro-economic
yield_10Y	10-year US sovereign bonds yields	Macro-economic
S&P500_ret	S&P 500 quarterly returns	Macro-economic

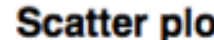
The second step is to split the dataset into training, validation and test sets. The rule of thumb followed here, is to divide the set into 60% for training (from year 2010 to 2014), 20% for validation (years 2015 and 2016) and 20% for testing (years 2017 and 2018).

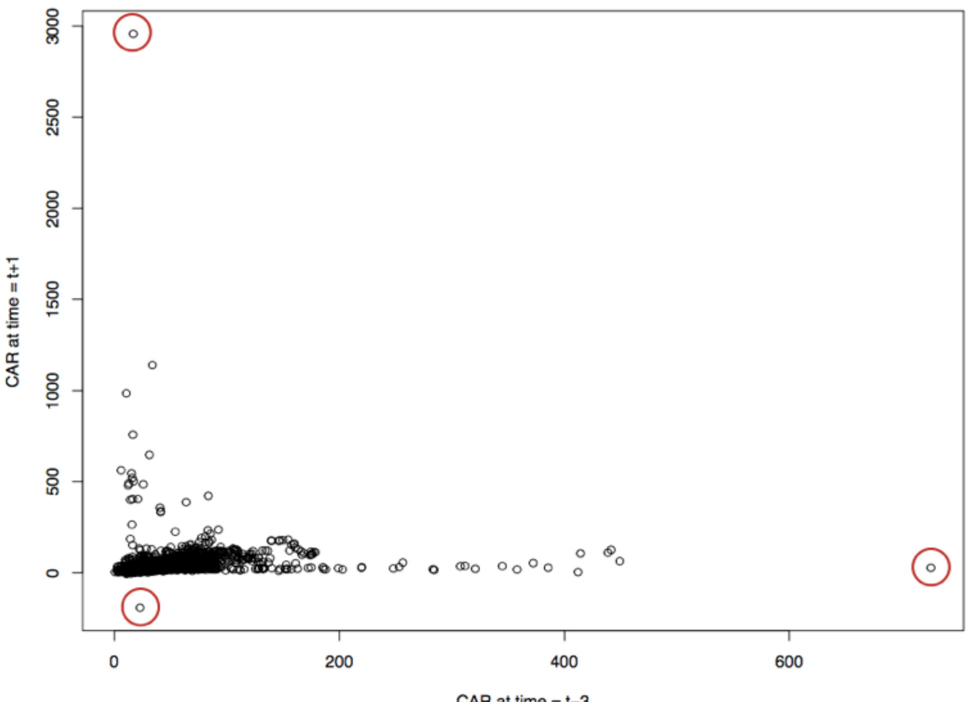
The following step when developing an ANN, is to carefully perform outliers' detection and data pre-processing in order to avoid premature saturation of the neurons and to improve the convergence speed.

It is not simple to decide which values can be considered as outliers. The dataset contains samples of both small and big banks. It is important to not exclude big banks who have extreme values for the financial variables, especially since they are systematically important banks and have to be monitored closely for ensuring the financial stability of the system. To find potential outliers in a dataset, one could check for missing or wrong values. However, having collected the data from two US governmental agencies, it can be safely assumed that the data went through careful inspection before being published. After these considerations, what can be done is to plot the scatter plot for each independent variable against the dependent variable and check if there are values far away from the others.

Figure 6.1 shows the scatter plot of the CAR ratio at time t-3 against the CAR ratio at time t+1. The three points circled in red are far away from the others and can be considered as outliers. Checking the performance of the three banks over time, it was found that they failed the next quarter after showing these extreme values. It is reasonable then to exclude these points from the dataset.

Figure 6.1. Scatter plot of CAR ratio at t-3 against CAR ratio at t+1





As for the data pre-processing, it is essential to scale the variables into a common scale since the variables have quite different ranges. There exist several methods of scaling. In this analysis, two of these methods are compared, namely standardization and robust scaling. The formula for standardization is as follows:

$$x_i' = \frac{x_i - \mu}{\sigma} \tag{6.3}$$

where μ is the mean and σ is the standard deviation.

The robust scaling can be computed as:

$$x_i' = \frac{x_i - Q_2}{Q_3 - Q_1} \tag{6.4}$$

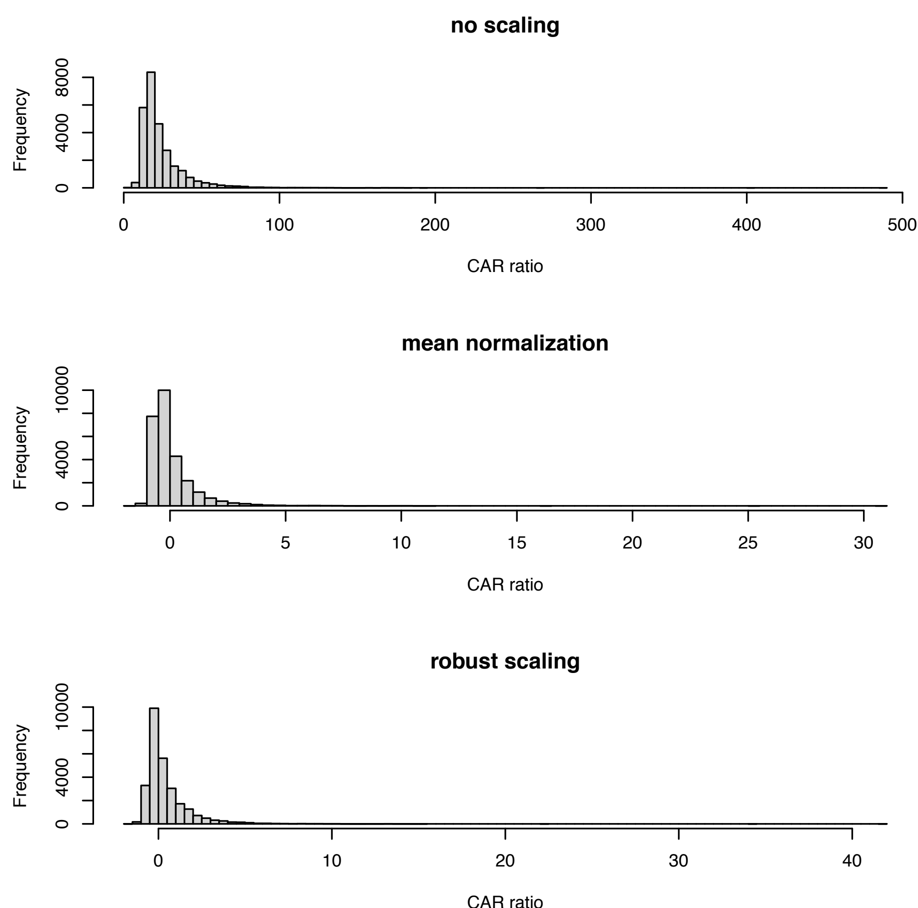
where Q_1, Q_2, Q_3 are the 25th, 50th and 75th percentiles.

It should be noted that the mean, standard deviation and percentiles should be estimated only on the training dataset and then applied also to the validation and test sets in order to avoid data leakage. If on the other hand, the estimates are computed on the overall dataset, they will carry also information about the

validation and test sets. The model will learn that information during training and it will result in an overestimated prediction accuracy.

In order to choose the optimal method, it is possible to plot the distribution of each variable and to examine how it changes after the two methods are applied. Figure 6.2 shows as an example the histogram of the CAR ratio at time t. It can be seen that both scaling technique do not change much the shape of distribution. However, the standardization method compresses the values into a smaller range and this could help to improve further the convergence speed. For this reason, the choice falls on the standardization method.

Figure 6.2. Histogram of CAR ratio at t for different scaling techniques



6.2 Model development and hyperparameters tuning

The problem in question is a regression problem, in which the one-year ahead CAR ratio is predicted as a function of the financial and macro-economic variables at time *t*, *t-1*, *t-2* and *t-3*. In order to do so, an initial ANN characterized by a simple architecture and a common choice of parameters is developed.

As an initial trial, a feedforward neural network with only one hidden layer composed by 55 hidden neurons is developed. For the choice of the number of hidden neurons, a simple rule of thumb is followed, so that the number of hidden neurons equals to two thirds of the input neurons. The activation function chosen for the hidden nodes is ReLu, which is a non-linear activation function and it computes $f(z) = \max(0, z)$. ReLu is a popular choice and it is typically used in conjunction with the dropout regularization technique to improve the model performance. The optimization method chosen is the stochastic gradient descent, as it converges faster than the classic gradient descent. The final output node computes and optimizes the root mean squared error (RMSE) loss during the backward propagation and it outputs the forecasted value during the forward propagation. The other parameters are also set to common values, in particular the number of training epochs is set to 500, the batch size is set to 200, the learning rate is set to 1e-3, while the weight decay (L2 regularization coefficient) and the momentum are set to 0 for now.

It is important also to initialize properly the weights and the thresholds. For ReLu activations, a slight modification of Glorot and Bengio (2010) formula is used. The weights are initialized from a uniform distribution in the interval:

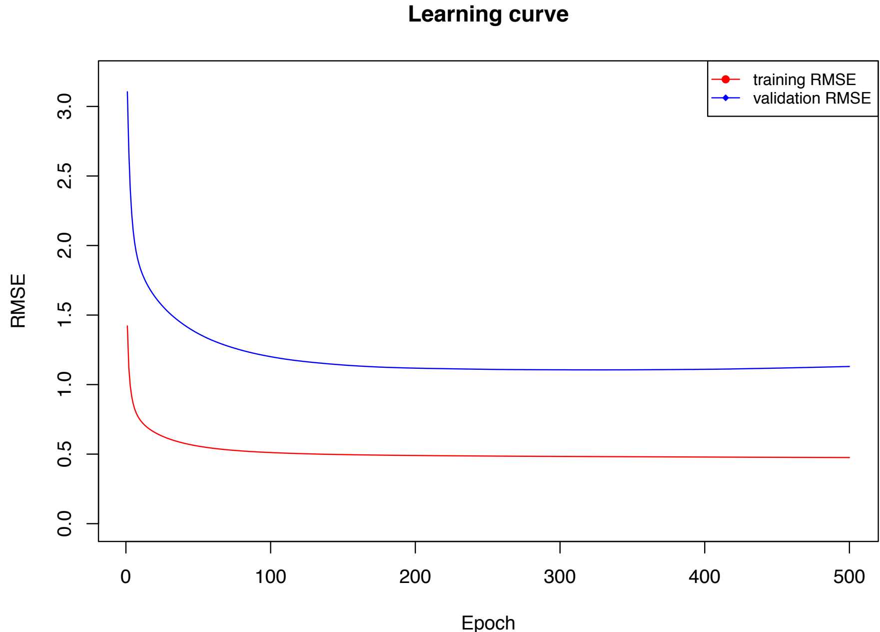
$$\left[-\sqrt{\frac{12}{n_{in}+n_{out}}}, +\sqrt{\frac{12}{n_{in}+n_{out}}}\right] \tag{6.5}$$

Where n_{in} is the number of neurons in the input layer and n_{out} is the number of neurons in the output layer. Having 80 input neurons and 1 output neuron, the initialization interval is $[-0.385, +0.385]$.

During the training process it is useful to plot the learning curve of the training and validation error over each epoch. This plot gives a first hint on the training

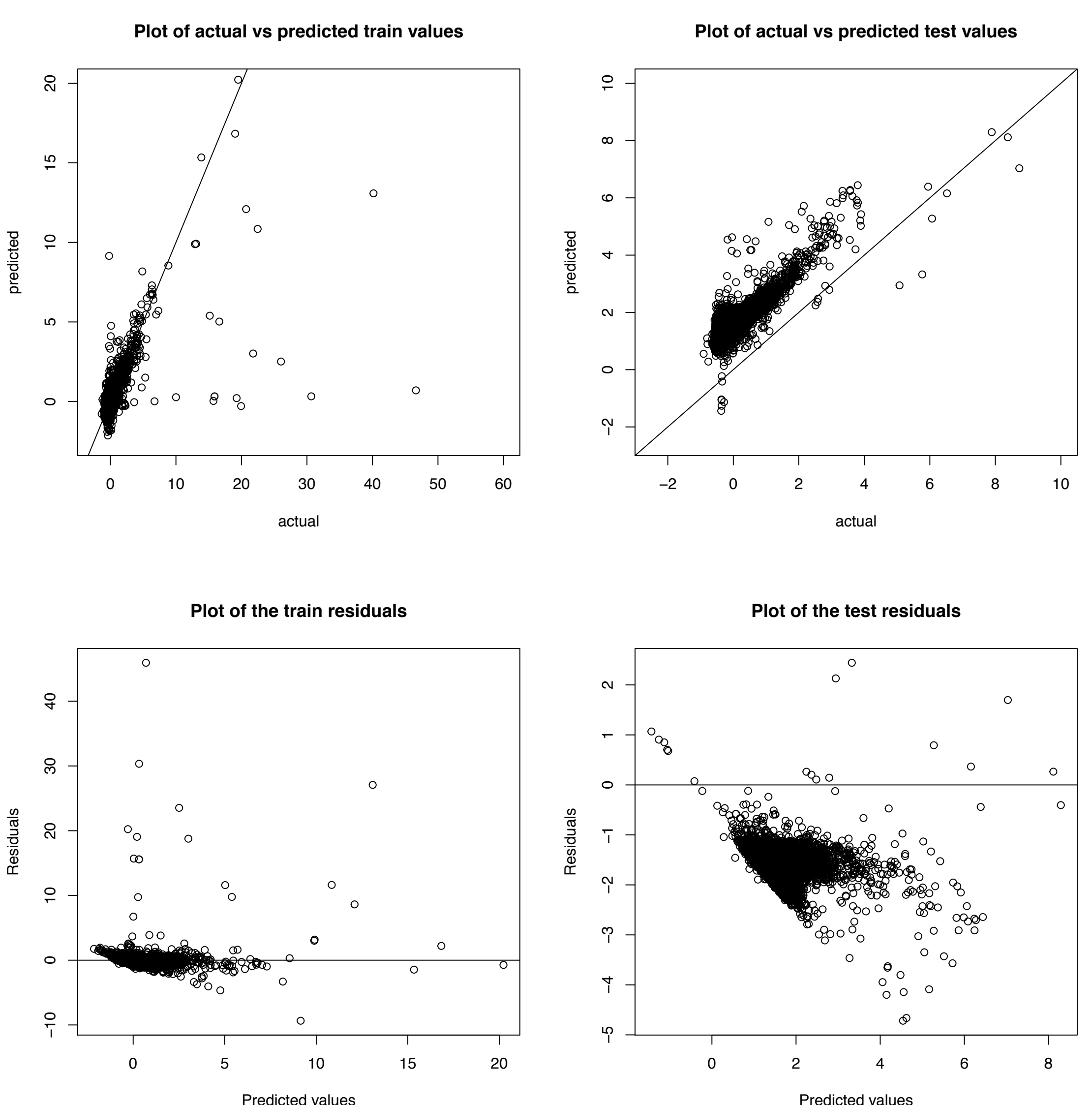
performance and if there are underfitting or overfitting issues. Figure 6.3 shows the plot of the learning curve of the initial model. Both the training and validation error are decreasing, however at the end of epoch 500 the validation error is much higher than training error, meaning that there is room for improvement in the performance.

Figure 6.3. Learning curve over epoch for the initial model



It is possible also to plot predicted against actual values and predicted values against residuals for both training and test sets to see how well the model fits the data. Figure 6.4 shows these plots for the initial model.

Figure 6.4. Actual by predicted and residual plots for training and test sets



From the actual against predicted plot, it can be seen that train values lie in a much larger range that test values, making the training set more difficult to learn than the validation set. The reason for this large difference is that the training data is composed by years in which the economic cycle is facing a recession, while the test data belongs to a more favorable phase. If the data is kept like that, in the following stages there will be issues with the validation error being lower than the training error. To solve this issue, a step back needs to be taken and additional outliers have to be removed from the training set. To decide how many values should be excluded, one can try comparing how many examples lie above several

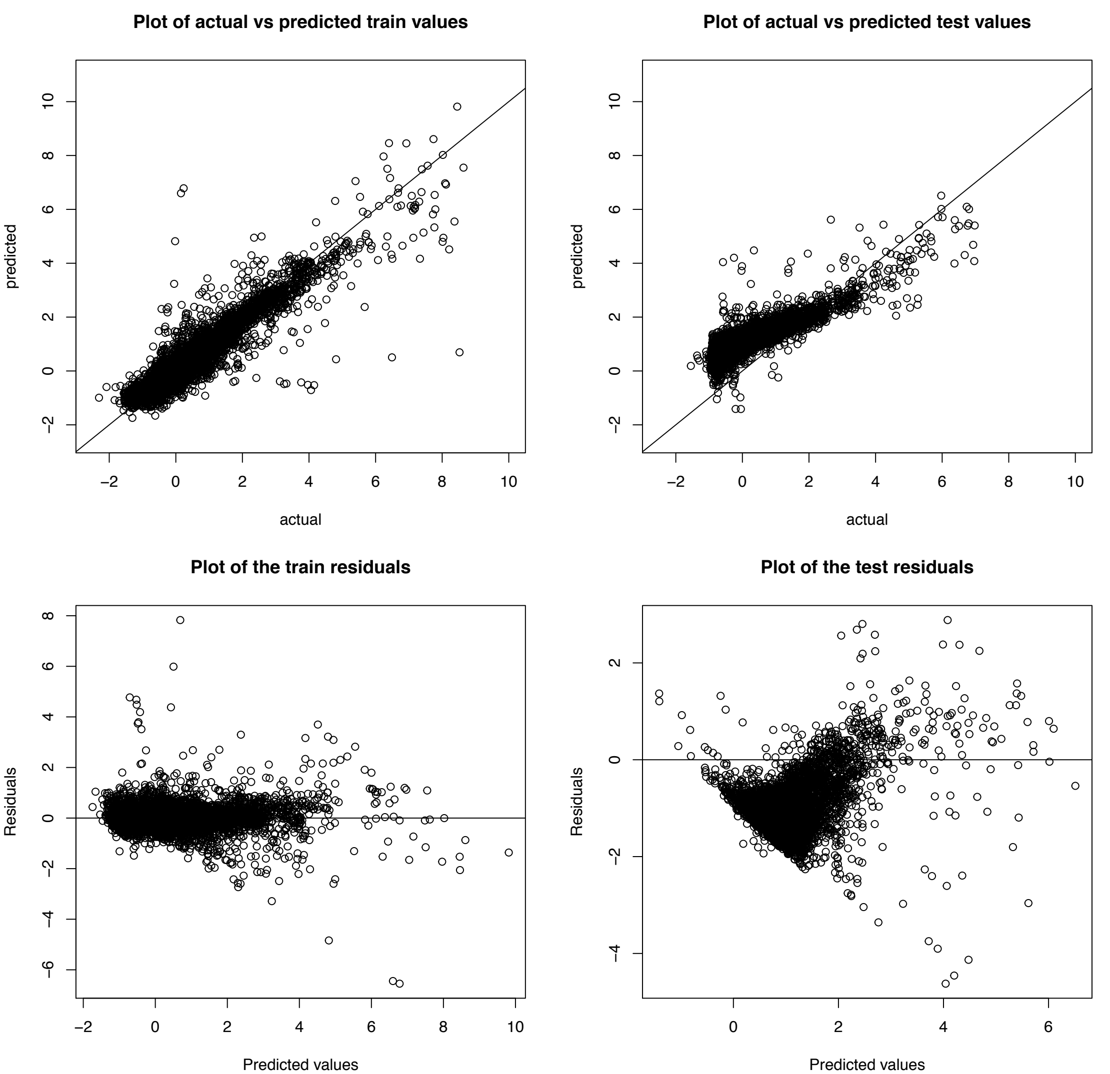
different thresholds. Table 6.2 represents a set of thresholds and the corresponding number of examples above that threshold.

Table 6.2. Thresholds with corresponding number of examples

Threshold	3	4	5	6	7	8	9	10
N° examples	163	80	46	36	25	23	20	20

The threshold of 5 seems a good compromise between diminishing the range of the training data distribution and not excluding too many values. After these values are removed and data is rescaled, the model can be run again on the new data. As can be seen from Figure 6.5, the two plots of actual against predicted values of the new dataset lie on a similar range.

Figure 6.5. Actual by predicted and residual plots for new training and test sets



The residuals plot, however, has still an unusual shape due to the fact that the model is too simple to explain the relationship between the variables. The choice of a proper architecture and the tuning of the hyperparameters will address this issue.

As an additional diagnostic measure, another type of learning curve, showing the relationship between the training set size and the RMSE, can be also represented. Ideally the training error should increase up to a certain threshold, while the validation error should decrease until the two curves nearly converge. When there are only a few examples, the training error will be low. Adding new examples will increase the training error, but at the same time the model will start to generalize better and the validation error will decrease. At a certain point adding new examples will not lead to any increase in the training data nor a decrease in the validation data, meaning that the model cannot be improved anymore by adding new data.

This type of plot can be used to check if sufficient data was collected, but also to diagnose underfitting or overfitting issues. If the two errors are high and the validation curve flattens too soon, this can be a sign of underfitting. If on the other hand, the validation curve flattens too slowly and the final gap between the validation and training error is too large, then there may be a problem of overfitting.

Figure 6.6. Learning curve for the training set size of the cleaned model

Figure 6.6 shows the learning curve for the training set size of the cleaned model. From this plot, it can be seen that the validation error decreases slowly and after 10000 examples it does not diminish much more, while the training error does not increase, indicating that the sample size is sufficient. The final gap between the two errors is still large, so that there is still room for improvement in model performance.

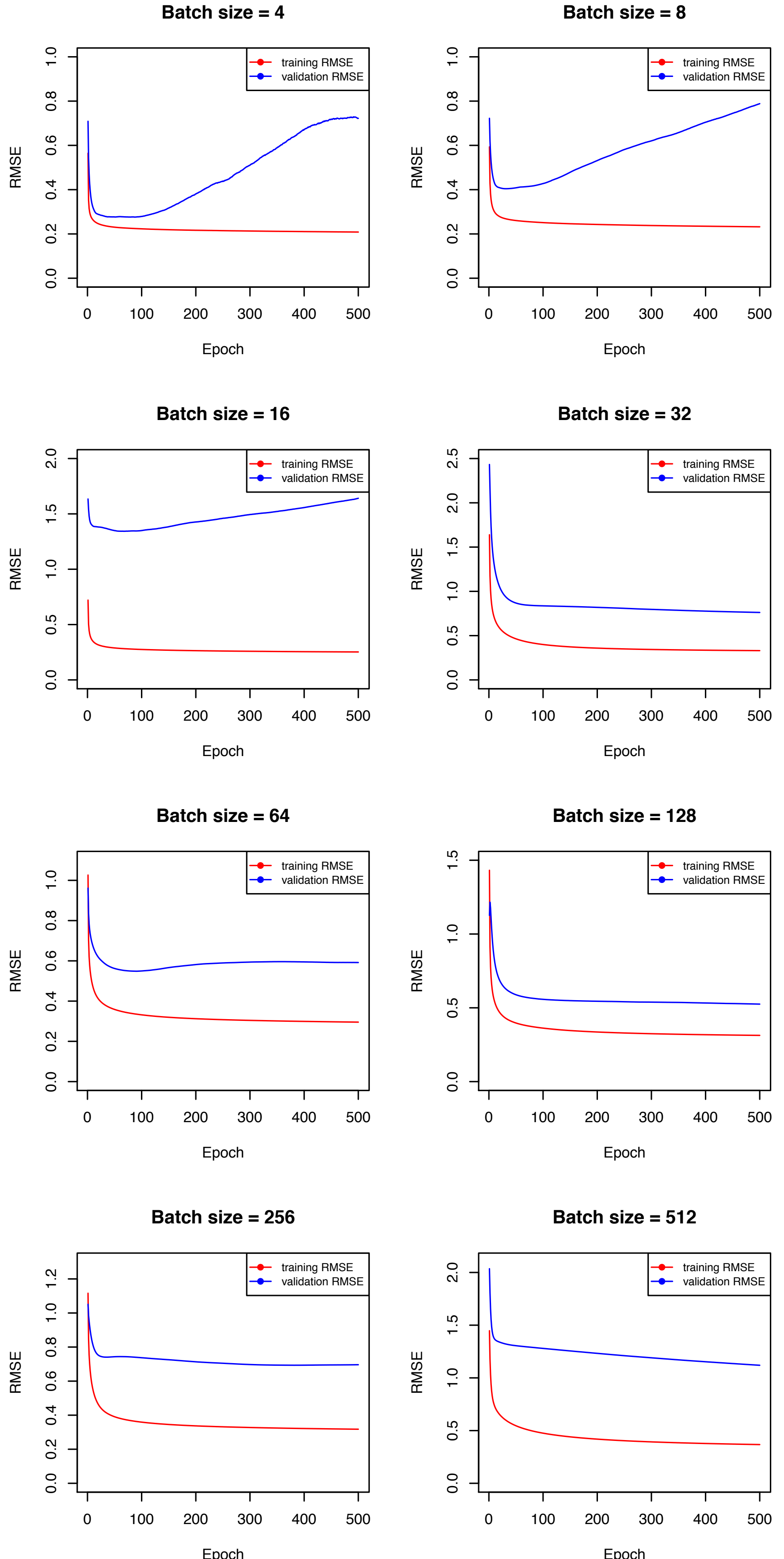
The next step is to tune the hyperparameters and to choose a proper architecture. The first parameter to be tuned is the batch size, which controls how many training examples are fed into the network before it updates its internal parameters. The smaller the batch size, the noisier and less accurate will be the estimate of the gradient. Figure 6.7 shows how the learning curve differs for a common set of batch sizes. The values used to construct this plot are summarized in table 6.3.

Table 6.3. Training and validation RMSE for each batch size

Batch size	Training RMSE	Validation RMSE
4	0.2808884	0.7646563
8	0.2909414	0.8103767
16	0.2980046	1.6498182
32	0.3394789	0.7629813
64	0.3185743	0.5967911
128	0.3278236	0.5278244
256	0.3302692	0.6987758
512	0.3695705	1.1214295

The batch sizes of 4, 8 and 16 are characterized by an increasing validation curve, which is a sign of overfitting. For the other batch sizes, both curves decrease smoothly. For the batch size of 128, the validation error is both closer to the training error and lower with respect to the other graphs. When plotting a learning curve to choose the optimal value for the hyperparameter, the lowest validation RMSE is used to select the best model. As a measure for the actual generalization capacity of the chosen model, one should consider instead the test error. In this case the test error is equal to 1.0995523.

Figure 6.7. Learning curve for different batch sizes



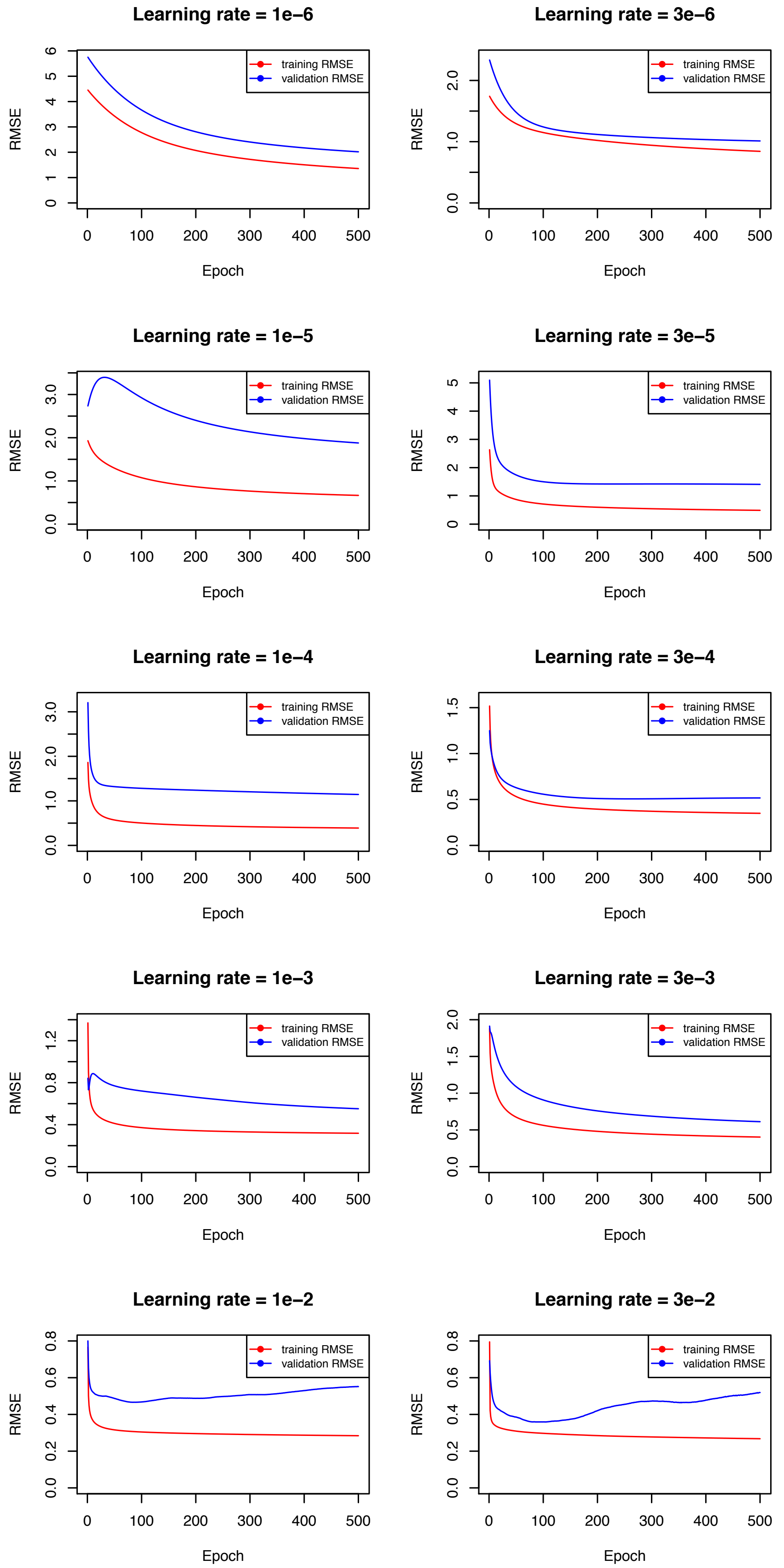
The next parameter that can be adjusted is the learning rate, which controls the step size that is taken by the learning algorithm during each update. The smaller is the learning rate, the more iterations the model will need to converge to a solution. Figure 6.8 shows how the learning curve changes for different values of the learning rate. The values used to construct this plot are summarized in table 6.4.

Table 6.4. Training and validation RMSE for each learning rate

Learning rate	Training RMSE	Validation RMSE
1e-6	1.3605340	2.0211911
3e-6	0.8477103	1.0196106
1e-5	0.6738588	1.8796661
3e-5	0.4996461	1.4100378
1e-4	0.3989935	1.1459491
3e-4	0.3627621	0.5192023
1e-3	0.3336582	0.5541194
3e-3	0.4142433	0.6158273
1e-2	0.2973949	0.5540108
3e-2	0.2752589	0.5207355

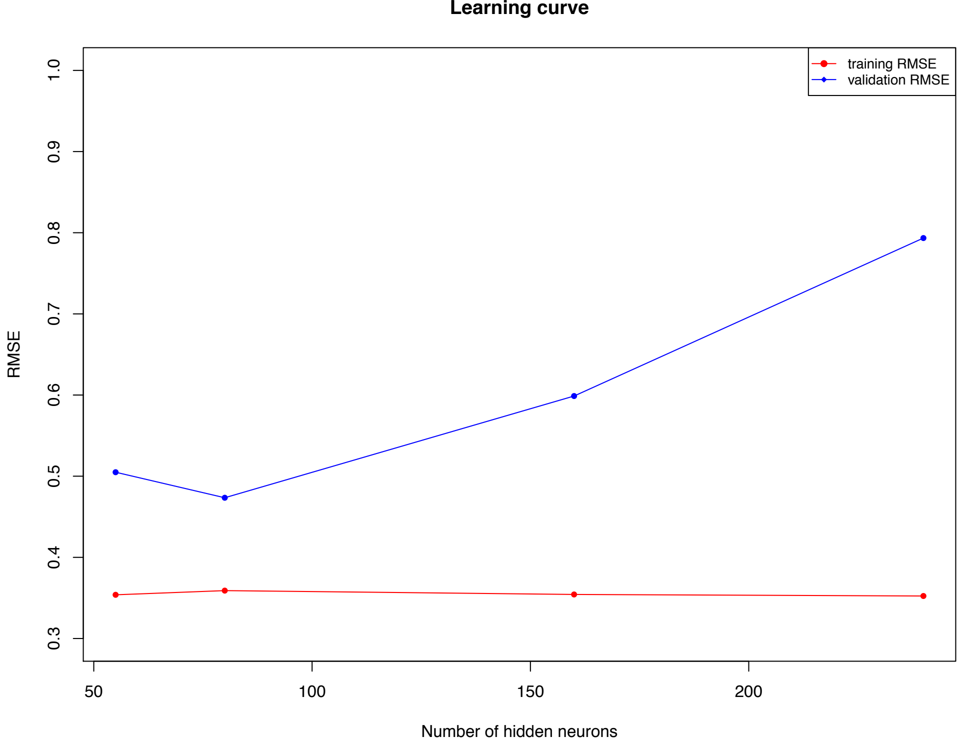
An alfa equal to 3e-4 gives the lowest validation error and can be considered as a good choice for the model. The test error for the optimal model is equal to 0.6018634.

Figure 6.8. Learning curve for different learning rates



The next step is to find the optimal number of hidden neurons. Figure 6.9 shows the plot of a learning curve representing the RMSE against the number of hidden neurons. It can be seen that the optimal number of neurons leading to the lowest error is 80. The test error for this model is equal to 0.5245490.

Figure 6.9. Learning curve for different numbers of hidden neurons



The values used to construct this plot are summarized in table 6.5.

Table 6.5. Training and validation RMSE for different number of hidden neurons

N° of hidden neurons	Training RMSE	Validation RMSE
55	0.3538492	0.5049096
80	0.3590196	0.4734417
160	0.3542810	0.5987814
240	0.3523830	0.7934415

In this case, since the number of hidden neurons is equal to the number of inputs, only one hidden layer is required. If more hidden neurons were to be selected, it would be possible also to divide the nodes into more hidden layers (having the same number of nodes in each layer) and to choose the architecture with the lowest error.

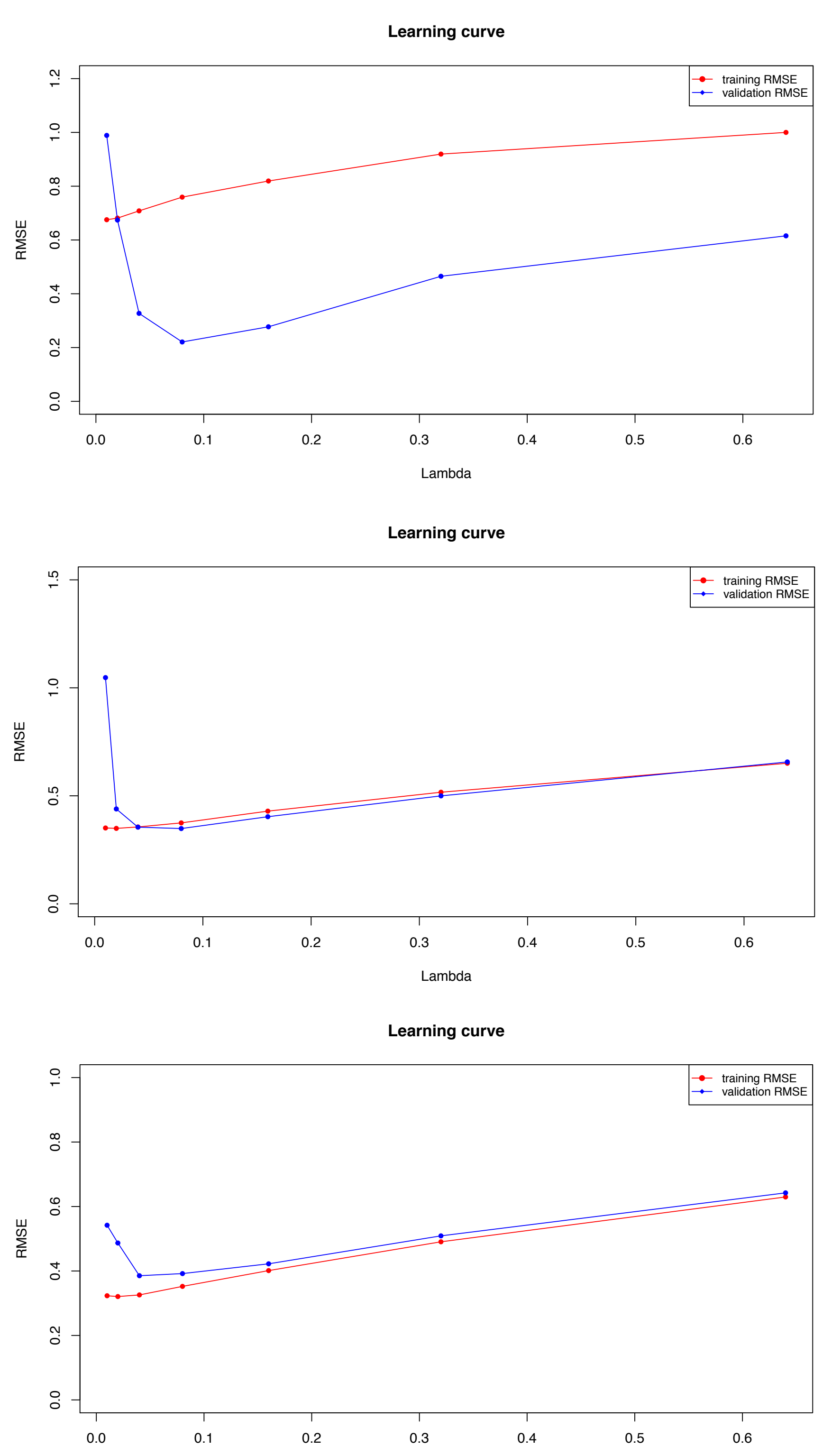
The subsequent stage is to select a proper weight decay, which is essential to solve the overfitting issue that ANNs often encounter. The plot of the learning curve for different values of lambda can help choosing the optimal value. The learning curve for lambda after the additional outliers' removal has still the validation curve lying below the training curve in some points. This is due to how the dataset was divided into the three subsets. The validation set is slightly easier with respect to the training set.

To solve this issue, a possible solution is to mix the years instead of the initial simple spit. The years 2010, 2012, 2014, 2016 and 2018 are placed into the training set, the years 2011 and 2015 into the validation set, while the years 2013 and 2017 into the test set. The tuning of the weight decay and the remaining parameters can be done with the new subsets. Figure 6.10 shows the three learning curves for the regularization parameter: the first graph is for the model with no additional outliers' removal, the second plot is for the model after the additional outliers were removed and the third is for model having the mixed years. From these plots, it is clear how dealing with outliers and dividing properly the three subsets can impact the learning curve for lambda. The validation curve is far lower than the training curve for the first model, for the second it is only slightly lower and for the third the validation error is always greater than the training error. The optimal lambda after mixing the years has a value of 0.04. The test RMSE for this model is equal to 0.3524313. The values used to construct the plot for the mixed data are summarized in table 6.6.

Table 6.6. Training and validation RMSE for each lambda

Lambda	Training RMSE	Validation RMSE
0.01	0.3231508	0.5418966
0.02	0.3206690	0.4868148
0.04	0.3257047	0.3852540
0.08	0.3521838	0.3919100
0.16	0.4011927	0.4222282
0.32	0.4905156	0.5089262
0.64	0.6296357	0.6423207

Figure 6.10. Learning curve for weight decay (initial, cleaned and mixed years models)



Another parameter that can be adjusted is momentum. This parameter acts directly on the learning algorithm by adding an inertia to the past direction of the gradient and by speeding up the search process. To choose the optimal value for the momentum, the model is run for three common values and the errors are saved for each model. Table 6.7 shows the training and validation error for each value of the momentum. The value equal to 0.9 gives the lowest validation RMSE and is the optimal choice. The test RMSE is equal to 0.3179015 in this case.

Table 6.7. Training and validation RMSE for each momentum value

Momentum	Training RMSE	Validation RMSE
0.5	0.3178722	0.3548670
0.9	0.3158370	0.3376132
0.99	0.3175656	0.3412118

In addition to all the hyperparameters that were previously tuned, it is also possible to add a dropout operation to the hidden layer, which randomly and temporarily deactivates a prespecified percentage of the hidden neurons. Dropout can be used as a regularization technique in place of L2 regularization or it can be used in conjunction with it. One should try both the alternatives and choose the model which gives the best performance.

At first, the model is run with dropout only, by inserting the dropout operation after the hidden layer, by keeping the momentum equal to 0.9 and by setting the weight decay to 0. This is done for several values of the dropout probability ranging from 0.1 to 0.8 and the errors are saved in a table. Looking at Table 6.8, the optimal dropout probability is 0.8. The validation error however is greater than the validation error of the model with weight decay equal to 0.04. For this problem, dropout alone does not improve the performance.

Next, the model is run with the optimal weight decay of 0.04, the momentum equal to 0.9 and the dropout operation for the same range of values as before. Table 6.9 shows the training and validation error for each value of dropout probability. Here the optimal probability is 0.1, however the validation error is still higher than the

one of the model with only L2 regularization. For this problem, combining dropout together with L2 regularization does not improve the performance of the model.

The final model selected during the hyperparameters tuning phase is the one with weight decay equal to 0.04 and momentum equal to 0.9. For this model the learning curve for the epochs is plotted again, as shown in Figure 6.11. Compared to the learning curve of the initial trial, there is a great improvement and the validation error converges far better to the training error.

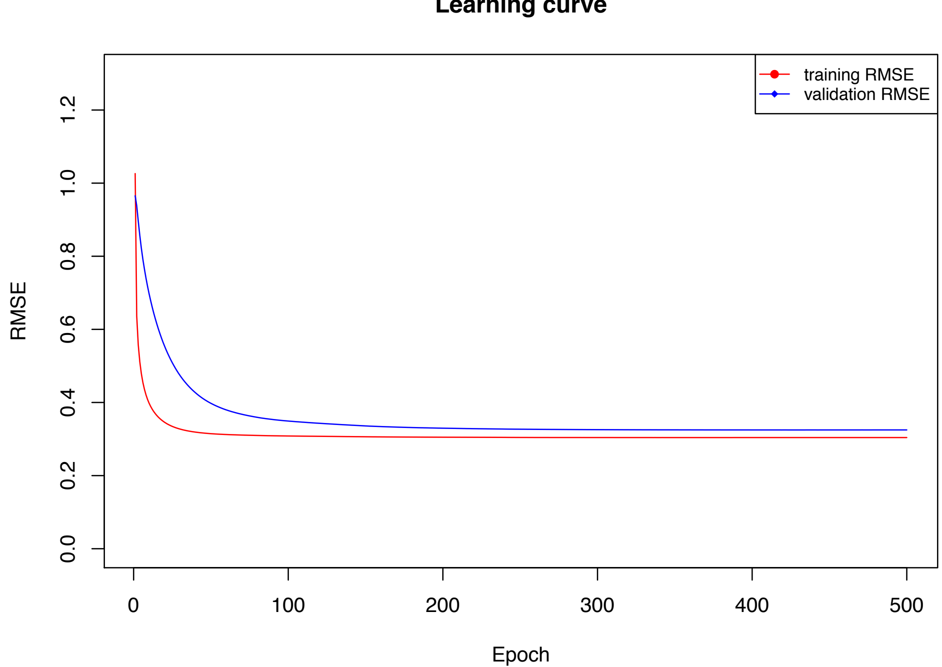
Table 6.8. Training and validation RMSE for dropout only

Dropout probability	Training RMSE	Validation RMSE
0.1	0.2980202	0.3930612
0.2	0.2988269	0.3797138
0.3	0.3014141	0.3802287
0.4	0.2951145	0.3896060
0.5	0.2986773	0.3877924
0.6	0.2981306	0.3895990
0.7	0.2974126	0.4112488
0.8	0.2977595	0.3475960

Table 6.9. Training and validation RMSE for dropout and L2 regularization

Dropout probability	Training RMSE	Validation RMSE
0.1	0.3169915	0.3379163
0.2	0.3169869	0.3379721
0.3	0.3180193	0.3394906
0.4	0.3168687	0.3381536
0.5	0.3178455	0.3391089
0.6	0.3174190	0.3387054
0.7	0.3168625	0.3385629
0.8	0.3172123	0.3385263

Figure 6.11. Learning curve over epoch for the final model



Looking at the plot of predicted against actual values and predicted values against residuals in Figure 6.12, it can be also seen a great improvement. For the test values, the points are much closer to the fitted line and the test residuals do not have an unusual shape anymore.

In order to synthetize the progress made by tuning the hyperparameters, a final table summarizing the training, validation and test error, after each parameter has been adjusted, is presented. As can be seen from Table 6.10, there is a visible improvement in the model performance after each parameter is adjusted. The final model could be still improved by using more advanced techniques. Nevertheless, using a simple hyperparameters' tuning, a satisfactory predictive performance can be reached.

A final consideration should be done also on the prediction of the CAR ratio and how the model results should be interpreted by a regulator. As was pointed out in Chapter 4, the CAR ratio measures if a bank has sufficient capital into reserve to withstand a certain amount of losses due to an external shock. The minimum

requirement for the CAR ratio is 8%. The higher is the CAR ratio, the more likely is the bank to withstand the shock. The lower it is, the higher is the risk of failure.

The standardized requirement of 8% can be computed in order to interpret the results. The CAR ratio at time *t+1* has a mean equal to 23.79% and a standard deviation equal to 13.80%. The standardized minimum is then equal to -1.14%. A regulator will have to monitor more carefully the banks having a standardized prediction close to -1.14% or even below. Those banks will be more likely to not have sufficient risk coverage and they could trigger the collapse of the entire economy in the case that they are important nodes in the system. Based on the results of the model, the regulators could decide for which banks it is necessary to take preemptive measures that will reduce their level of risk.

Having this in mind, the plot of predicted against actual values can be seen under a new light. The points lying on the left side of the plot around the value of -1 represent the banks at a higher risk that need careful monitoring. From Figure 6.12, it can be seen that the points below -1 are overestimated by the model and this would be a serious issue, if the model had to be used in practice. The banks with a CAR ratio below -1 are at higher risk than it is forecasted and a wrong estimation of the actual risk incurred will be made by looking only at these predictions. That is why one should interpret the results with caution and if possible, the final model can be still improved to obtain more precise estimates for low values of the standardized CAR ratio.

Figure 6.12. Actual by predicted and residual plots for training and test sets of the final model

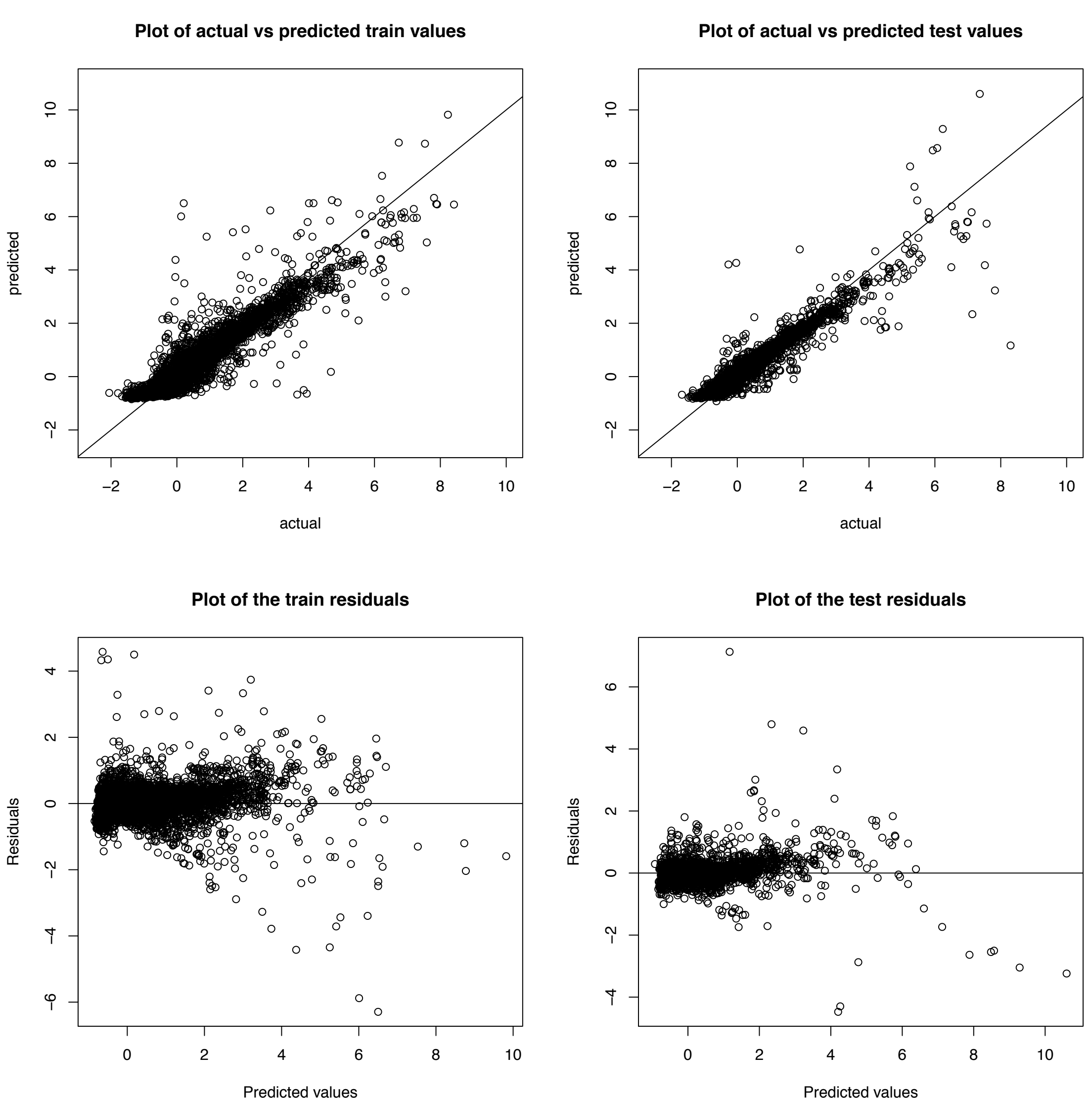


Table 6.10. Summary of training, validation and test RMSE for each tuning

Type of tuning	Training RMSE	Validation RMSE	Test RMSE
No tuning	0.3348050	0.7411400	1.1869470
Batch size = 128	0.3278236	0.5278244	1.0995523
Learning rate = 3e-4	0.3627621	0.5192023	0.6018634
N° of hidden neurons = 80	0.3590196	0.4734417	0.5245490
Weight decay = 0.04	0.3257047	0.3852540	0.3524313
Momentum = 0.9	0.3158370	0.3376132	0.3179015

Conclusions and closing remarks

With respect to the final case study analysis, some important takeaways have to pointed out before making any final considerations.

First, ANNs need a large amount of data and variables in order to be able to generalize. When trying to solve a financial problem using neural networks, it is not simple to find sufficient data. One should first look for possible sources of data and verify if there are sufficient examples for the problem in question. Depending on the data availability, the variables should be selected thereafter.

Second, it is essential to pre-process the dataset, to divide properly the three subsets and to remove all the possible outliers. As the results of the case study show, processing adequately the dataset will have a major impact on the model performance and also on the soundness of the estimated errors.

Third, one should not stop at the common choice for the architecture and the hyperparameters of the network, but instead he should choose a rigorous method for tuning them to improve the results.

Considering the lessons learnt, it is clear that one should study and thoroughly understand the theoretical notions behind artificial neural networks to obtain reasonable results and to be able to interpret them correctly.

More broadly, the focus of this thesis has been to explore the theoretical foundations relative to artificial neural networks and subsequently how they can be applied for solving the financial problems that can be encountered by banks and regulatory authorities. ANNs could be employed to identify the potential risks and weaknesses faced by banks through non-linear linkages. The insights drawn from these instruments could be used in addition to the supervisory toolkit to identify troubled institutions or potentially important systemic nodes.

It should be stressed once again that one should possess a sound knowledge of the theory and the properties of the model, along with a clear understanding of the financial problem in question in order to employ correctly ANNs. If the model is used

alone, it will produce outputs that are unstructured or difficult to interpret. Consequently, a certain degree of subjectivity and a critical mind are needed when analyzing and interpreting the results, especially when dealing with prediction problems.

In conclusion, artificial neural networks and deep learning techniques could be valid instruments to be added into the supervisory toolkit for performing stress tests in order to get novel insights on the relations between the variables and identify other sources of vulnerability.

Bibliography

[Abraham, 2005] Abraham, A. (2005), *Artificial Neural Networks*, In Handbook of Measuring System Design (eds P.H. Sydenham and R. Thorn).

[Acemoglu et al., 2015] Acemoglu, D., Ozdaglar, A. E., Tahbaz-Salehi, A. (2015), *Systemic Risk and Stability in Financial Networks*, American Economic Review, American Economic Association, vol. 105(2), pp. 564–608.

[Adrian et al., 2020] Adrian, T., Morsink, J., Schumacher, L. B. (2020), *Stress Testing at the IMF*, IMF Departmental Papers, Policy Papers 20/04, International Monetary Fund.

[Allen and Babus, 2008] Allen, F., Babus, A. (2008), *Networks in Finance*, The Network Challenge: Strategy, Profit, and Risk in an Interlinked World.

[Al-Shayea et al., 2010] Al-Shayea, Q., El-Refae, G. A., El-Itter, S. F. (2010), *Neural Networks in Bank Insolvency Prediction*, International Journal of Computer Science and Network Security, 10 (5), pp. 240–245.

[Anderson and McNeil, 1992] Anderson, D., McNeil, G. (1992), *Artificial Neural Networks Technology*, ADACS (Data & Analysis Center for Software) State-of-the Art Report.

[Angelini et al., 2008] Angelini, E., Tollo, G. D., Roli, A. (2008), *A neural network approach for credit risk evaluation*, The Quarterly Review of Economics and Finance, 48, pp. 733–755.

[Atiya, 2001] Atiya, A.F. (2001), *Bankruptcy prediction for credit risk using neural networks: A survey and new results*, IEEE transactions on neural networks, 12 (4), pp. 929–935.

[Aydin and Cavdar, 2015] Aydin, A. D., Cavdar, S. C. (2015), Prediction of Financial Crisis with Artificial Neural Network: An Empirical Analysis on Turkey, International Journal of Financial Research, 6 (4), pp. 36–45.

[Bahrammirzaee, 2010] Bahrammirzaee, A. (2010), *A comparative survey of artificial intelligence applications in finance: artificial neural networks, expert system and hybrid intelligent systems*, International Journal of Neural Computing & Application, 19, pp. 1165–1195.

[Basel Committee, 2004] Basel Committee (2004), *Principles for Sound Stress Testing Practices and Supervision*, Bank of International Settlements.

[Basel Committee, 2009] Basel Committee (2009), *International Convergence of Capital Measurement and Capital Standards*, consultative paper, Bank of International Settlements.

[Basel Committee, 2011] Basel Committee (2011), *Basel III: A global regulatory framework for more resilient banks and banking systems*, Bank for International Settlements.

[Basel Committee, 2014] Basel Committee (2014), *Basel III: the net stable funding ratio*, Bank for International Settlements.

[Basel Committee, 2016] Basel Committee (2016), *Leverage and Risk Weighted Capital Requirements*, Bank for International Settlements, Working Papers, 586.

[Basheer and Hajmeer, 2000] Basheer, I. A. and Hajmeer, M. (2000), *Artificial Neural Networks: Fundamentals, Computing, Design, and Application*, Journal of microbiological methods, 43, pp. 3-31.

[Cerchiello et al., 2017] Cerchiello, P., Nicola, G., Rönnqvist, S., Sarlin, P. (2017), *Deep learning bank distress from news and numerical financial data*, DEM Working Papers Series, 140, University of Pavia, Department of Economics and Management.

[Chatzis et al., 2018] Chatzis, S. P., Siakoulis, V., Petropoulos, A., Stavroulakis, E., Vlachogiannakis, N. (2018), *Forecasting stock market crisis events using deep and statistical machine learning techniques*, Expert Systems with Applications, 112, pp. 353–371.

[Čihák, 2007] Čihák, M. (2007), *Introduction to Applied Stress Testing*, IMF Working Papers 07/59, International Monetary Fund.

[Dahl et al., 2013] Dahl, G. E., Sainath, T. N., Hinton, G. E. (2013), *Improving deep neural networks for LVCSR using rectified linear units and dropout*, IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 8609-8613.

[Da Silva et al., 2017] Da Silva, I. N., Spatti, D. H., Flauzino, R. A., Liboni, L. H. B., dos Reis Alves, S. F. **(2017),** ***Artificial Neural Networks:*** *A Practical Course*, Springer International Publishing.

[De Mello and Ponti, 2018] De Mello, R. F., Ponti, M. A. (2018), ***Machine learning: A practical approach on the statistical learning theory*,** Springer International Publishing.

[EBA, 2018] European Banking Authority (2018), *Final report on guidelines on institutions' stress testing*, EBA/GL/2018/04.

[ECB, 2006] European Central Bank (2006), Financial Stability Review, pp. 147-154.

[Enoch et al., 2013] Enoch, C., Everaert, L., Tressel, T., Zhou, J. (2013), *From Fragmentation to Financial Integration in Europe*, International Monetary Fund.

[Fioramanti, 2008] Fioramanti, M. (2008), *Predicting Sovereign Debt Crises Using Artificial Neural Networks: A Comparative Approach*, Journal of Financial Stability, 4, pp. 149–164.

[Gai and Kapadia, 2010] Gai, P., Kapadia, S. (2010), *Contagion in financial networks*, Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 466, pp. 2401–2423.

[Gai et al., 2011] Gai, P., Haldane, A.G., Kapadia, S. (2011), *Complexity, concentration and contagion*, Journal of Monetary Economics, Elsevier, vol. 58(5), pp. 453–470.

[Glorot and Bengio, 2010] Glorot, X., Bengio, Y. (2010), *Understanding the difficulty of training deep feedforward neural networks*, Proceedings of AISTATS 2010, vol. 9, pp. 249– 256.

[Graves, 2012] Graves A. (2012), *Supervised Sequence Labelling with Recurrent Neural Networks*, Studies in Computational Intelligence, Springer, Berlin, Heidelberg.

[Haldane, 2013] Haldane, A. (2013), *Rethinking the financial network*, Fragile Stabilität – stabile Fragilität, Springer VS, Wiesbaden, pp. 243–278.

[Haldane and May, 2011] Haldane, A. G., May, R. M. (2011), *Systemic risk in banking ecosystems*, Nature, 469, pp. 351–355.

[Heaton et al., 2016] Heaton, J. B., Polson, N. G., Witte, J.H. (2016), *Deep Learning in Finance*, Applied Stochastic Models in Business and Industry, 33 (1), pp. 3–12.

[Hodnett et al., 2019] Hodnett, M., Wiley, J. F., Liu, Y. H., Maldonado, P. (2019), *Deep Learning with R for Beginners: Design neural network models in R 3.5 using TensorFlow, Keras, and MXNet*, Packt Publishing Ltd, pp. 97-130.

[IMF, 2013] International Monetary Fund (2013), *Financial Crises: Explanations, Types, and Implications*, WP/13/28.

[IMF, FSB and BIS, 2009] International Monetary Fund, Financial Stability Board and Bank for International Settlements (2009), *Guidance to Assess the Systemic Importance of Financial Institutions, Markets and Instruments: Initial Considerations*, Report to the G-20 Finance Ministers and Central Bank Governors, Washington, DC and Basel.

[Ioffe and Szegedy, 2015] Ioffe, S., Szegedy, C. (2015), *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, ICML'15:

Proceedings of the 32nd International Conference on International Conference on Machine Learning, Volume 37, pp. 448–456.

[Khashman, 2010] Khashman, A. (2010), *Neural networks for credit risk evaluation: Investigation of different neural models and learning schemes*, Expert Systems with Applications, 37, pp. 6233–6239.

[Kumar and Ravi, 2007] Kumar, P. R., Ravi, V. (2007), *Bankruptcy prediction in banks and firms via statistical and intelligent techniques - A review*, European Journal of Operational Research, 180, pp. 1–28.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., Hinton, G. (2015), *Deep Learning,* Nature 521 (7553), pp. 436-444.

[Lipton et al., 2015] Lipton, Z. C., Berkowitz, J., Elkan, C. (2015), *A Critical Review of Recurrent Neural Networks for Sequence Learning*, ArXiv, abs/1506.00019.

[Liu et al., 2017] Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., & Alsaadi, F.E. (2017), *A survey of deep neural network architectures and their applications*, Neurocomputing, 234, pp. 11-26.

[Mai et al., 2019] Mai, F., Tian, S., Lee, C., & Ma, L. (2019). *Deep learning models for bankruptcy prediction using textual disclosures*, European Journal of Operational Research, 274, pp. 743–758.

[Maier et al., 2010] Maier, H. R., Jain, A., Dandy, G. C., Sudheer, K. P. (2010), *Methods used for the development of neural networks for the prediction of water resource variables in river systems: Current status and future directions*, Environmental Modelling & Software, 25, pp. 891-909.

[Nielsen, 2015] Nielsen, M. A. (2015), *Neural Networks and Deep Learning*, Determination Press, available at http://neuralnetworksanddeeplearning.com/

[Nik et al., 2016] Nik, P. A., Jusoh, M., Shaari, A. H., and Sarmdi, T. (2016), *Predicting the probability of financial crisis in emerging countries using an early warning system: artificial neural network*, Journal of Economic Cooperation & Development, 37 (1), pp. 25–40.

[Pacelli and Azzollini, 2011] Pacelli, V., Azzollini, M. (2011), *An Artificial Neural Network Approach for Credit Risk Management*, Journal of Intelligent Learning Systems and Applications, 3, pp. 103–112.

[Petropoulos et al., 2018] Petropoulos, A., Siakoulis, V., Stavroulakis, E., Klamargias, A. (2018), *A robust machine learning approach for credit risk analysis of large loan level datasets using deep learning and extreme gradient boosting*, IFC Bulletins chapters, 49.

[Petropoulos et al., 2019] Petropoulos, A., Siakoulis, V., Vlachogiannakis, N. E., Stavroulakis E. (2019), *Deep-Stress: A deep learning approach for dynamic balance*

sheet stress testing, 8th Annual Research Workshop - The future of stress tests in the banking sector – approaches, governance and methodologies, Paris, 2019.

[Petropoulos et al., 2020] Petropoulos, A., Siakoulis, V., Stavroulakis, E., Vlachogiannakis, N. E. (2020), *Predicting bank insolvencies using machine learning techniques*, International Journal of Forecasting, https://doi.org/10.1016/j.ijforecast.2019.11.005.

[Quagliarello, 2009] Quagliarello, M. (2009), *Stress testing the banking system: Methodologies and applications*, Cambridge: Cambridge University Press.

[Santurkar et al., 2018] Santurkar, S., Tsipras, D., Ilyas, A., Madry, A. (2018), *How Does Batch Normalization Help Optimization?*, NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems, pp. 2488–2498.

[Schuermann, 2014] Schuermann, T. (2014), *Stress Testing Banks*, International Journal of Forecasting, vol. 30 (3), pp. 717–728.

[Sengupta et al., 2020] Sengupta, S., Basak, S., Saikia, P., Paul, S., Tsalavoutis, V., Atiah, F.D., Ravi, V., Peters, R.A. (2020), *A review of deep learning with special emphasis on architectures, applications and recent trends*, Knowledge-Based Systems, 105596, https://doi.org/10.1016/j.knosys.2020.105596

[West, 2000] West, D. (2000), *Neural network credit scoring models*, Computers & Operations Research, 27, pp. 1131–1152.

[Wu et al., 2014] Wu, W., Dandy, G. C., Maier, H. R. (2014), *Protocol for developing ANN models and its application to the assessment of the quality of the ANN model development process in drinking water quality modelling*, Environmental Modelling & Software, 54, pp. 108-127.

[Yu et al., 2008] Yu, L., Wang, S., Lai, K. K. (2008), *Credit risk assessment with a multistage neural network ensemble learning approach*, Expert Systems with Applications, 34, pp. 1434–1444.

Appendix A
A list of the codes for the case study

The present appendix contains the codes that were produced for the case study analysis of the final chapter. The Apache MXNet toolbox in R is employed for the development of the ANN and the tuning of the hyperparameters [14].

A.1 Outliers' detection and scaling choice

```
#### outliers_det.R ####

# this script contains the code to make a scatterplot for the
# detection of outliers

# load the raw dataset
rm(list = ls())
library(readxl)
data <- read_excel("bank_data_final.xlsx")
df = data[,c(3:83)]
df[is.na(df)] <- 0
df_matrix <- matrix(as.matrix(df), ncol = ncol(df), dimnames = NULL)

# plot the outliers with a scatter plot
plot(df_matrix[,1],df_matrix[,81], ylab = "CAR at time = t+1", xlab =
"net_loan")
plot(df_matrix[,4],df_matrix[,81], ylab = "CAR at time = t+1", xlab =
"yield_ea")
plot(df_matrix[,5],df_matrix[,81], ylab = "CAR at time = t+1", xlab =
"fundc_ea")
plot(df_matrix[,7],df_matrix[,81], ylab = "CAR at time = t+1", xlab =
"CAR")
plot(df_matrix[,16],df_matrix[,81], ylab = "CAR at time = t+1", xlab =
"fundc_ea_1")
plot(df_matrix[,40],df_matrix[,81], ylab = "CAR at time = t+1", xlab =
"CAR at time = t-3")

#### scaling_choice.R ####

# this script contains the code to choose the best scaling method

### load the data without the outliers ###
rm(list = ls())
library(readxl)
data <- read_excel("bank_data_no_outl.xlsx")

df = data[,c(3:83)]
```

[14] The MXNet framework is available at https://mxnet.apache.org/

```
df[is.na(df)] <- 0
df_final <- matrix(as.matrix(df), ncol = ncol(df), dimnames = NULL)


### scale variables with the two techniques ###

# standardization
tr_mean <- apply(df_final[1:16023,], 2, mean) # computed for training
set only
tr_sd <- apply(df_final[1:16023,], 2, sd)

df_stand = matrix(, nrow = dim(df_final)[1], ncol = dim(df_final)[2])
for (i in 1:81) {
  df_stand[,i] = (df_final[,i]- tr_mean[i])/tr_sd[i]
}

# robust scaling
tr_Q1 <- apply(df_final[1:16023,], 2, quantile, probs = 0.25)
tr_Q2 <- apply(df_final[1:16023,], 2, quantile, probs = 0.5)
tr_Q3 <- apply(df_final[1:16023,], 2, quantile, probs = 0.75)

df_robust = matrix(, nrow = dim(df_final)[1], ncol = dim(df_final)[2])
for (i in 1:81) {
  df_robust[,i] = (df_final[,i]- tr_Q2[i])/(tr_Q3[i]-tr_Q1[i])
}

### plot histograms for comparison ###

pdf('plot_hist_100.pdf', width=6, height=8)
par(mfrow=c(3,1))
hist(df_final[,1], xlab = "net loans", breaks = 100, main = "no
scaling")
hist(df_stand[,1], xlab = "net loans", breaks = 100, main =
"standardization")
hist(df_robust[,1], xlab = "net loans", breaks = 100, main = "robust
scaling")

par(mfrow=c(3,1))
hist(df_final[,8], xlab = "total assets", breaks = 100, main = "no
scaling")
hist(df_stand[,8], xlab = "total assets", breaks = 100, main =
"standardization")
hist(df_robust[,8], xlab = "total assets", breaks = 100, main =
"robust scaling")

par(mfrow=c(3,1))
hist(df_final[,4], xlab = "yield on earning assets", breaks = 100,
main = "no scaling")
hist(df_stand[,4], xlab = "yield on earning assets", breaks = 100,
main = "standardization")
hist(df_robust[,4], xlab = "yield on earning assets", breaks = 100,
main = "robust scaling")

par(mfrow=c(3,1))
hist(df_final[,7], xlab = "CAR ratio", breaks = 100, main = "no
scaling")
hist(df_stand[,7], xlab = "CAR ratio", breaks = 100, main =
"standardization")
hist(df_robust[,7], xlab = "CAR ratio", breaks = 100, main = "robust
scaling")

par(mfrow=c(3,1))
```

```
hist(df_final[,81], xlab = "CAR ratio at time T+1", breaks = 100, main
= "no scaling")
hist(df_stand[,81], xlab = "CAR ratio at time T+1", breaks = 100, main
= "standardization")
hist(df_robust[,81], xlab = "CAR ratio at time T+1", breaks = 100,
main = "robust scaling")
dev.off()
```

A.2 Initial trial

```
#### intial_trial.R ####

# this script contains the code for the initial trial with parameters
# set equal to common values for the dataset with additional outliers
# removal

rm(list = ls())
library(readxl)
require(mxnet)

###### load the dataset ######

data <- read_excel("bank_data_no_outl_2.xlsx")

###### pre-processing of the data ######

# extract specific variables
df = data[,c(3:83)]
df[is.na(df)] <- 0 # impute missing values as 0

# scale the variables with standardization
df_final <- matrix(as.matrix(df), ncol = ncol(df), dimnames = NULL)

# compute for the train set only
tr_mean <- apply(df_final[1:15985,], 2, mean)
tr_sd <- apply(df_final[1:15985,], 2, sd)

# apply to whole dataset
df_norm = matrix(, nrow = dim(df_final)[1], ncol = dim(df_final)[2])
for (i in 1:81) {
df_norm[,i] = (df_final[,i]- tr_mean[i])/tr_sd[i]
}


###### partition dataset into train, validation and test sets ######

# partition the dataset into 60% (years 2010-2014),
# 20% (years 2015-2016) and 20% (years 2017-2018)
data_train = df_norm[1:15985,]
data_val = df_norm[15986:22064,]
data_test = df_norm[22065:27398,]

data_train.x = data_train[,-81] # predictors
data_train.y = data_train[,81] # response variable

data_val.x = data_val[,-81]
data_val.y = data_val[,81]

data_test.x = data_test[,-81]
```

```
data_test.y = data_test[,81]


###### create network architecture ######

data <- mx.symbol.Variable("data")

# FIRST HIDDEN LAYER - 55 neurons
fc1 <- mx.symbol.FullyConnected(data, name = "fc1", num_hidden=55)
act1 <- mx.symbol.Activation(fc1, name ="sigmoid1", act_type = "relu")
# output layer
fc_out <- mx.symbol.FullyConnected(act1, name="fc_out", num_hidden=1)
lro <-  mx.symbol.LinearRegressionOutput(fc_out)


###### train the model ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500

logger <- mx.metric.logger$new()
model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
            y = data_train.y, ctx=devices, array.layout = "rowmajor",
            eval.data=list(data=data_val.x, label=data_val.y),
            num.round = n_epoch, array.batch.size = 200,
            learning.rate = 1e-3, momentum = 0, wd = 0,
            eval.metric = mx.metric.rmse, initializer =
            mx.init.uniform(0.385),epoch.end.callback =
            mx.callback.log.train.metric(1,logger))


# save RMSE metrics (train + evaluation)
RMSE_train = logger$train
RMSE_eval = logger$eval

# plot the RMSE over the epochs
pdf('plot_lear_curve_initial_trial.pdf', width = 8, height = 6)
par(mfrow=c(1,1))
plot(RMSE_train,type = "l", col = "red", ylim = c(0,2.2),
     xlab ="Epoch", ylab ="RMSE")
lines(RMSE_eval,type = "l", col = "blue")
title(main ="Learning curve")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,18), lty=1:1)
dev.off()

# compute predictions and residuals for the train and validation data
y_pred_train <- predict(model_dnn, data_train.x)
summary(as.vector(y_pred_train))
resid_train = data_train.y - y_pred_train
sprintf("train RMSE = %f", sqrt(mean((resid_train)^2)))

y_pred_val <- predict(model_dnn, data_val.x)
summary(as.vector(y_pred_val))
resid_val = data_val.y - y_pred_val
sprintf("validation RMSE = %f", sqrt(mean((resid_val)^2)))

# compute predictions and residuals for the test data
y_pred_test <- predict(model_dnn, data_test.x)
summary(as.vector(y_pred_test))
resid_test = data_test.y - y_pred_test
sprintf("test RMSE = %f", sqrt(mean((resid_test)^2)))
```

```
###### plot of fitted against actual values and residuals ######

pdf('plot_pred_act_initial_trial.pdf', width = 12, height = 6)
par(mfrow=c(1,2))
plot(data_train.y, y_pred_train[1,],
     xlab = "actual", ylab = "predicted", xlim = c(-2.5,10), ylim =
c(-2.5,11),
     main = "Plot of actual vs predicted train values")
abline(a=0,b=1)
plot(data_test.y, y_pred_test[1,],
     xlab = "actual", ylab = "predicted",xlim = c(-2.5,10), ylim = c(-
2.5,11),
     main = "Plot of actual vs predicted test values")
abline(a=0,b=1)
dev.off()

pdf('plot_resid_pred_initial_trial.pdf', width = 12, height = 6)
par(mfrow=c(1,2))
plot(y_pred_train, resid_train,
     ylab = "Residuals", xlab = "Predicted values",
     main = "Plot of the train residuals")
abline(0, 0)
plot(y_pred_test, resid_test,
     ylab = "Residuals", xlab = "Predicted values",
     main = "Plot of the test residuals")
abline(0, 0)
dev.off()

###### convert weights and thresholds into a matrix ######
FC1_w = as.matrix(model_dnn$arg.params$fc1_weight)
FC1_b = as.array(model_dnn$arg.params$fc1_bias)
FC_out_w = as.matrix(model_dnn$arg.params$fc_out_weight)
FC_out_b = as.array(model_dnn$arg.params$fc_out_bias)


###### save the working environment ######

save.image(file = 'initial_trial.RData')
load('initial_trial.RData') # to load the environment later
```

A.3 Learning curve for the training set size

```
#### LC_train_size.R ####

# this script contains the code for creating the learning curve for
# different values of training set size

# the initial part of the script (in which the dataset is loaded and
# partitioned and the network architecture is created) is the same as
# in initial_trial.R, so it is omitted for simplicity


###### train the different model with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
```

```
m_train = c(2,100,500,1000,1500,2000,5000,10000, dim(data_train.x)[1])

RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()
for (i in m_train) {
  train.x_new = data_train.x[1:i,]
  train.y_new = data_train.y[1:i]
  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = train.x_new,
              y = train.y_new, ctx=devices, array.layout = "rowmajor",
              eval.data=list(data=data_val.x,
                             label=data_val.y),
              num.round = n_epoch, array.batch.size = 200,
              learning.rate = 1e-3, momentum = 0, wd = 0,
              eval.metric = mx.metric.rmse,
              epoch.end.callback = mx.callback.log.train.metric(1,
            logger),initializer = mx.init.uniform(0.385))
  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
  RMSE_eval = logger$eval
  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])

  # make predictions and save all results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)

  tr_pred_list[[match(i,m_train)]] = y_pred_train
  v_pred_list[[match(i,m_train)]] = y_pred_val
  tst_pred_list[[match(i,m_train)]] = y_pred_test

  models_list[[match(i,m_train)]] = model_dnn
  RMSE_list[[match(i,m_train)]] = logger

  rm(logger)
  rm(model_dnn)
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
```

```
  rm(err_tst)
  }


# plot the RMSE over the training set size
pdf('LC_train_size_plot.pdf', width = 10, height = 6)
plot(m_train,err_train,type = "o", col = "red", pch = 20, ylim =
c(0,3.5),
     xlab ="Training set size", ylab ="RMSE")
lines(m_train,err_eval,type = "o",pch = 20, col = "blue")
title(main ="Learning curve")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,18), lty=1:1)
dev.off()
```

A.4 Learning curve for the batch size

```
#### LC_batch_size.R ####

# this script contains the code for creating the learning curve for
# different values of batch size

# the initial part of the script (in which the dataset is loaded and
# partitioned and the network architecture is created) is the same as
# in initial_trial.R, so it is omitted for simplicity

###### train the different models with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
batch_s= c(4,8,16,32,64,128,256,512)


RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()

for (i in batch_s) {
  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
             y=data_train.y, ctx=devices, array.layout = "rowmajor",
             eval.data=list(data=data_val.x,
                            label=data_val.y),
             num.round = n_epoch, array.batch.size = batch_s,
             learning.rate = 1e-3, momentum = 0, wd = 0,
             eval.metric = mx.metric.rmse,
             epoch.end.callback = mx.callback.log.train.metric(1,
             logger), initializer = mx.init.uniform(0.385))

  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
```

```
  RMSE_eval = logger$eval

  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])

  # make predictions and save all results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)

  tr_pred_list[[match(i,batch_s)]] = y_pred_train
  v_pred_list[[match(i,batch_s)]] = y_pred_val
  tst_pred_list[[match(i,batch_s)]] = y_pred_test

  models_list[[match(i,batch_s)]] = model_dnn
  RMSE_list[[match(i,batch_s)]] = logger

  rm(logger)
  rm(model_dnn)
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
  rm(err_tst)

}


# plot the RMSE over epoch for each batch size
pdf('LC_batch_size_plot.pdf', width = 6, height = 12)
par(mfrow=c(4,2))
plot(RMSE_list[[1]]$train,type = "l", col = "red",
     ylim = c(0,1), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[1]]$eval,type = "l", col = "blue")
title(main ="Batch size = 4")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[2]]$train,type = "l", col = "red",
     ylim = c(0,1), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[2]]$eval,type = "l", col = "blue")
title(main ="Batch size = 8")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[3]]$train,type = "l", col = "red",
     ylim = c(0,2), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[3]]$eval,type = "l", col = "blue")
title(main ="Batch size = 16")
```

```
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[4]]$train,type = "l", col = "red",
     ylim = c(0,2.5),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[4]]$eval,type = "l", col = "blue")
title(main ="Batch size = 32")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[5]]$train,type = "l", col = "red",
     ylim = c(0,1.1),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[5]]$eval,type = "l", col = "blue")
title(main ="Batch size = 64")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[6]]$train,type = "l", col = "red",
     ylim = c(0,1.5),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[6]]$eval,type = "l", col = "blue")
title(main ="Batch size = 128")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[7]]$train,type = "l", col = "red",
     ylim = c(0,1.3), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[7]]$eval,type = "l", col = "blue")
title(main ="Batch size = 256")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[8]]$train,type = "l", col = "red",
     ylim = c(0,2.1),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[8]]$eval,type = "l", col = "blue")
title(main ="Batch size = 512")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

dev.off()

# save the RMSE values in a table
batches <- c("batch_4", "batch_8", "batch_16","batch_32",
             "batch_64", "batch_128", "batch_256", "batch_512")
RMSE_true_table <- data.frame("batch_size"= batches,
                              "train_RMSE"= tr_err_true,
                              "validation_RMSE" = v_err_true)
print(RMSE_true_table)
```

A.5 Learning curve for the learning rate

```
#### LC_alfa.R ####

# this script contains the code for creating the learning curve for
# different values of the learning rate (alfa)

# the initial part of the script (in which the dataset is loaded and
# partitioned and the network architecture is created) is the same as
# in initial_trial.R, so it is omitted for simplicity
```

```
###### train the different models with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
batch_s = 128
alfa = c(1e-6,3e-6,1e-5,3e-5,1e-4,3e-4,1e-3,3e-3,1e-2,3e-2)

RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()

for (i in alfa) {
  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
            y=data_train.y, ctx=devices, array.layout = "rowmajor",
            eval.data=list(data=data_val.x,
                           label=data_val.y),
            num.round = n_epoch, array.batch.size = batch_s,
            learning.rate = i, momentum = 0, wd = 0,
            eval.metric = mx.metric.rmse,
            epoch.end.callback = mx.callback.log.train.metric(1,
            logger), initializer = mx.init.uniform(0.385))

  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
  RMSE_eval = logger$eval

  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])

  # make predictions and save all results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)

  tr_pred_list[[match(i,alfa)]] = y_pred_train
  v_pred_list[[match(i,alfa)]] = y_pred_val
  tst_pred_list[[match(i,alfa)]] = y_pred_test

  models_list[[match(i,alfa)]] = model_dnn
  RMSE_list[[match(i,alfa)]] = logger

  rm(logger)
  rm(model_dnn)
```

```
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
  rm(err_tst)

}

# plot the RMSE over epoch for each alfa
pdf('LC_alfa.pdf', width = 6, height = 12)
par(mfrow=c(5,2))
plot(RMSE_list[[1]]$train,type = "l", col = "red",
     ylim = c(0,5.8), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[1]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 1e-6")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[2]]$train,type = "l", col = "red",
     ylim = c(0,2.4), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[2]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 3e-6")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[3]]$train,type = "l", col = "red",
     ylim = c(0,3.4), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[3]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 1e-5")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[4]]$train,type = "l", col = "red",
     ylim = c(0,5.2),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[4]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 3e-5")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[5]]$train,type = "l", col = "red",
     ylim = c(0,3.3),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[5]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 1e-4")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[6]]$train,type = "l", col = "red",
     ylim = c(0,1.6),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[6]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 3e-4")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[7]]$train,type = "l", col = "red",
     ylim = c(0,1.4), xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[7]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 1e-3")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)
```

```
plot(RMSE_list[[8]]$train,type = "l", col = "red",
     ylim = c(0,2),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[8]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 3e-3")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[9]]$train,type = "l", col = "red",
     ylim = c(0,0.8),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[9]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 1e-2")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

plot(RMSE_list[[10]]$train,type = "l", col = "red",
     ylim = c(0,0.8),xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[10]]$eval,type = "l", col = "blue")
title(main ="Learning rate = 3e-2")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,19), lty=1:1)

dev.off()

# save the RMSE values in a table
RMSE_true_table <- data.frame("n_alfa"=alfa,
                              "train_RMSE"= tr_err_true,
                              "validation_RMSE" = v_err_true)
print(RMSE_true_table)
```

A.6 Learning curve for the number of hidden neurons

```
#### LC_n_hidden.R ####

# this script contains the code for creating the learning curve for
# different numbers of hidden neurons

# the initial part of the script (in which the dataset is loaded and
# partitioned and the network architecture is created) is the same as
# in initial_trial.R, so it is omitted for simplicity

###### train the different models with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
batch_s = 128
alfa = 3e-4
n_hid <- c(55,80,160,240)

RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()
```

```
for (i in n_hid) {
  fc1 <- mx.symbol.FullyConnected(data, name = "fc1", num_hidden = i)

  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
              y=data_train.y, ctx = devices,array.layout = "rowmajor",
              eval.data=list(data=data_val.x,
                             label=data_val.y),
              num.round = n_epoch, array.batch.size = batch_s,
              learning.rate = alfa, momentum = 0, wd = 0,
              eval.metric = mx.metric.rmse,
              epoch.end.callback = mx.callback.log.train.metric(1,
              logger), initializer = mx.init.uniform(0.385))

  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
  RMSE_eval = logger$eval

  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])

  # make predictions and save all results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)

  tr_pred_list[[match(i,n_hid)]] = y_pred_train
  v_pred_list[[match(i,n_hid)]] = y_pred_val
  tst_pred_list[[match(i,n_hid)]] = y_pred_test

  models_list[[match(i,n_hid)]] = model_dnn
  RMSE_list[[match(i,n_hid)]] = logger

  rm(logger)
  rm(model_dnn)
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
  rm(err_tst)

}


# plot the RMSE over the number of hidden neurons
pdf('LC_n_hidden.pdf', width = 10, height = 8)
```

```
plot(n_hid, tr_err_true,type = "o",  pch = 20, col = "red", ylim =
c(0.3,1),
     xlab ="Number of hidden neurons", ylab ="RMSE")
lines(n_hid, v_err_true,type = "o", pch = 20, col = "blue")
title(main ="Learning curve")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,18), lty=1:1)
dev.off()

# save the RMSE values in a table
RMSE_true_table <- data.frame("n_hidden"= n_hid,
                              "train_RMSE"= tr_err_true,
                              "validation_RMSE" = v_err_true)
print(RMSE_true_table)
```

A.7 Learning curve for the weight decay

```
#### LC_lambda_mix.R ####

# this script contains the code for creating the learning curve for
# different values of lambda having a mixed dataset

rm(list = ls())
library(readxl)
require(mxnet)

###### load the dataset ######

data <- read_excel("bank_data_no_outl_2.xlsx")

###### pre-processing of the data ######

# extract specific variables
df = data[,c(3:83)]
df[is.na(df)] <- 0 # impute missing values as 0

# scale the variables with standardization
df_final <- matrix(as.matrix(df), ncol = ncol(df), dimnames = NULL)

tr_mean <-
apply(df_final[c(1:3375,6638:9841,12921:15985,19058:22064,24810:27398)
,], 2, mean)
tr_sd <-
apply(df_final[c(1:3375,6638:9841,12921:15985,19058:22064,24810:27398)
,], 2, sd)

df_norm = matrix(, nrow = dim(df_final)[1], ncol = dim(df_final)[2])
for (i in 1:81) {
  df_norm[,i] = (df_final[,i]- tr_mean[i])/tr_sd[i]
}


###### partition dataset into train, validation and test sets ######

# partition the dataset into 60% (years 2010-2014),
# 20% (years 2015-2016) and 20% (years 2017-2018)
data_train =
df_norm[c(1:3375,6638:9841,12921:15985,19058:22064,24810:27398),]
data_val = df_norm[c(3376:6637,15986:19057),]
```

```
data_test = df_norm[c(9842:12920,22065:24809),]

data_train.x = data_train[,-81] # predictors
data_train.y = data_train[,81] # response variable

data_val.x = data_val[,-81]
data_val.y = data_val[,81]

data_test.x = data_test[,-81]
data_test.y = data_test[,81]


###### create network architecture ######

data <- mx.symbol.Variable("data")

# 1st hidden layer – 80 neurons
fc1 <- mx.symbol.FullyConnected(data, name = "fc1", num_hidden=80)
act1 <- mx.symbol.Activation(fc1, name ="relu1", act_type = "relu")
# output layer
fc_out <- mx.symbol.FullyConnected(act1, name="fc_out", num_hidden=1)
lro <-  mx.symbol.LinearRegressionOutput(fc_out)


###### train the different models with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
batch_s = 128
alfa = 3e-4
lambda <- c(0.01,0.02,0.04,0.08,0.16,0.32,0.64)

RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()

for (i in lambda) {
  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
              y=data_train.y, ctx=devices, array.layout = "rowmajor",
              eval.data=list(data=data_val.x,
                             label=data_val.y),
              num.round = n_epoch, array.batch.size = batch_s,
              learning.rate = alfa, momentum = 0, wd = i,
              eval.metric = mx.metric.rmse,
              epoch.end.callback = mx.callback.log.train.metric(1,
              logger), initializer = mx.init.uniform(0.385))

  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
  RMSE_eval = logger$eval

  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])
```

```
  # make predictions and save all results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)

  tr_pred_list[[match(i,lambda)]] = y_pred_train
  v_pred_list[[match(i,lambda)]] = y_pred_val
  tst_pred_list[[match(i,lambda)]] = y_pred_test

  models_list[[match(i,lambda)]] = model_dnn
  RMSE_list[[match(i,lambda)]] = logger

  rm(logger)
  rm(model_dnn)
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
  rm(err_tst)

}

# plot the RMSE over epoch
pdf('LC_lambda_mix.pdf', width = 10, height = 6)
par(mfrow=c(1,1))
plot(lambda, tr_err_true,type = "o",  pch = 20, col = "red", ylim =
c(0,1),
     xlab ="Lambda", ylab ="RMSE")
lines(lambda, v_err_true,type = "o", pch = 20, col = "blue")
title(main ="Learning curve")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.4, col = c("red","blue"), pch = c(19,18), lty=1:1)
dev.off()


# save the RMSE values in a table
RMSE_true_table <- data.frame("n_lambda"=lambda,
                              "train_RMSE"= tr_err_true,
                              "validation_RMSE" = v_err_true)
print(RMSE_true_table)
```

A.8 Choice of the momentum

```
#### mom_mix.R ####
```

```
# this script contains the code for creating the learning curve for
# different values of momentum

# the initial part of the script (in which the dataset is loaded and
# partitioned and the network architecture is created) is the same as
# in LC_lambda_mix.R, so it is omitted for simplicity

###### train the different models with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
batch_s = 128
alfa = 3e-4
lambda = 0.04
momnt <- c(0.5,0.9,0.99)


RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()

for (i in momnt) {
  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
            y=data_train.y, ctx=devices, array.layout = "rowmajor",
            eval.data=list(data=data_val.x,
                           label=data_val.y),
            num.round = n_epoch, array.batch.size = batch_s,
            learning.rate = alfa, momentum = i, wd = lambda,
            eval.metric = mx.metric.rmse,
            epoch.end.callback = mx.callback.log.train.metric(1,
            logger), initializer = mx.init.uniform(0.385))

  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
  RMSE_eval = logger$eval

  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])

  # make predictions and save all results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)
```

```
  tr_pred_list[[match(i,momnt)]] = y_pred_train
  v_pred_list[[match(i,momnt)]] = y_pred_val
  tst_pred_list[[match(i,momnt)]] = y_pred_test

  models_list[[match(i,momnt)]] = model_dnn
  RMSE_list[[match(i,momnt)]] = logger

  rm(logger)
  rm(model_dnn)
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
  rm(err_tst)

}


# save the RMSE values in a table
RMSE_true_table <- data.frame("momentum"= momnt,
                              "train_RMSE"= tr_err_true,
                              "validation_RMSE" = v_err_true)
print(RMSE_true_table)
```

A.9 Choice of the dropout probability

```
#### only_drop_mix.R ####

# this script contains the code for creating the learning curve for
# different values of dropout probability with lambda = 0


rm(list = ls())
library(readxl)
require(mxnet)

###### load the dataset ######

data <- read_excel("bank_data_no_outl_2.xlsx")

###### pre-processing of the data ######

# extract specific variables
df = data[,c(3:83)]
df[is.na(df)] <- 0 # impute missing values as 0

# scale the variables with standardization
df_final <- matrix(as.matrix(df), ncol = ncol(df), dimnames = NULL)

tr_mean <-
apply(df_final[c(1:3375,6638:9841,12921:15985,19058:22064,24810:27398)
,], 2, mean)
tr_sd <-
apply(df_final[c(1:3375,6638:9841,12921:15985,19058:22064,24810:27398)
,], 2, sd)
```

```
df_norm = matrix(, nrow = dim(df_final)[1], ncol = dim(df_final)[2])
for (i in 1:81) {
  df_norm[,i] = (df_final[,i]- tr_mean[i])/tr_sd[i]
}


###### partition the data into train, validation and test sets ######

# partition the dataset into 60% (years 2010-2014),
# 20% (years 2015-2016) and 20% (years 2017-2018)
data_train =
df_norm[c(1:3375,6638:9841,12921:15985,19058:22064,24810:27398),]
data_val = df_norm[c(3376:6637,15986:19057),]
data_test = df_norm[c(9842:12920,22065:24809),]

data_train.x = data_train[,-81] # predictors
data_train.y = data_train[,81] # response variable

data_val.x = data_val[,-81]
data_val.y = data_val[,81]

data_test.x = data_test[,-81]
data_test.y = data_test[,81]


###### create network architecture ######

data <- mx.symbol.Variable("data")

# 1st hidden layer – 80 neurons
fc1 <- mx.symbol.FullyConnected(data, name = "fc1", num_hidden=80)
act1 <- mx.symbol.Activation(fc1, name ="relu1", act_type = "relu")
drop1 <- mx.symbol.Dropout(data = act1, p=0.1)
# output layer
fc_out <- mx.symbol.FullyConnected(drop1, name="fc_out", num_hidden=1)
lro <-  mx.symbol.LinearRegressionOutput(fc_out)


###### train the different models with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
batch_s = 128
alfa = 3e-4
lambda = 0
momnt = 0.9
dropout <- c(0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8)


RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()

for (i in dropout) {
```

```
  drop1 <- mx.symbol.Dropout(data = act1, p=i)
  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
              y=data_train.y, ctx=devices, array.layout = "rowmajor",
              eval.data=list(data=data_val.x,
                             label=data_val.y),
              num.round = n_epoch, array.batch.size = batch_s,
              learning.rate = alfa, momentum = momnt, wd = lambda,
              eval.metric = mx.metric.rmse,
              epoch.end.callback = mx.callback.log.train.metric(1,
              logger), initializer = mx.init.uniform(0.385))

  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
  RMSE_eval = logger$eval

  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])

  # make predictions and save results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)

  tr_pred_list[[match(i,dropout)]] = y_pred_train
  v_pred_list[[match(i,dropout)]] = y_pred_val
  tst_pred_list[[match(i,dropout)]] = y_pred_test

  models_list[[match(i,dropout)]] = model_dnn
  RMSE_list[[match(i,dropout)]] = logger

  rm(logger)
  rm(model_dnn)
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
  rm(err_tst)

}


# save the RMSE values in a table
RMSE_true_table <- data.frame("dropout_prob"= dropout,
                              "train_RMSE"= tr_err_true,
                              "validation_RMSE" = v_err_true)
print(RMSE_true_table)
```

A.10 Choice of the dropout probability with L2 regularization

```
#### drop_mix.R ####

# this script contains the code for creating the learning curve for
# different values of dropout probability with lambda = 0.04

# the initial part of the script (in which the dataset is loaded and
# partitioned and the network architecture is created) is the same as
# in only_drop_mix.R, so it is omitted for simplicity

###### train the different models with a for cycle ######

devices <- mx.cpu()
mx.set.seed(0)
n_epoch = 500
batch_s = 128
alfa = 3e-4
lambda = 0.04
momnt = 0.9
dropout <- c(0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8)


RMSE_list = list()
models_list = list()
tr_pred_list = list()
v_pred_list = list()
tst_pred_list = list()
tr_err_true <- c()
v_err_true <- c()
tst_err_true <- c()
err_train <- c()
err_eval <- c()

for (i in dropout) {
  drop1 <- mx.symbol.Dropout(data = act1, p=i)
  logger <- mx.metric.logger$new()
  model_dnn = mx.model.FeedForward.create(lro, X = data_train.x,
              y=data_train.y, ctx=devices, array.layout = "rowmajor",
              eval.data=list(data=data_val.x,
                             label=data_val.y),
              num.round = n_epoch, array.batch.size = batch_s,
              learning.rate = alfa, momentum = momnt, wd = lambda,
              eval.metric = mx.metric.rmse,
              epoch.end.callback = mx.callback.log.train.metric(1,
              logger), initializer = mx.init.uniform(0.385))

  # save RMSE final metrics (train + evaluation)
  RMSE_train = logger$train
  RMSE_eval = logger$eval

  err_train = c(err_train, RMSE_train[n_epoch])
  err_eval = c(err_eval, RMSE_eval[n_epoch])

  # make predictions and save results
  y_pred_train <- predict(model_dnn, data_train.x, array.layout =
  "rowmajor")
  y_pred_val <- predict(model_dnn, data_val.x, array.layout =
  "rowmajor")
  y_pred_test <- predict(model_dnn, data_test.x, array.layout =
```

```
  "rowmajor")

  err_tr = sqrt(mean((data_train.y - y_pred_train)^2))
  err_v = sqrt(mean((data_val.y - y_pred_val)^2))
  err_tst = sqrt(mean((data_test.y - y_pred_test)^2))

  tr_err_true = c(tr_err_true, err_tr)
  v_err_true = c(v_err_true, err_v)
  tst_err_true = c(tst_err_true, err_tst)

  tr_pred_list[[match(i,dropout)]] = y_pred_train
  v_pred_list[[match(i,dropout)]] = y_pred_val
  tst_pred_list[[match(i,dropout)]] = y_pred_test

  models_list[[match(i,dropout)]] = model_dnn
  RMSE_list[[match(i,dropout)]] = logger

  rm(logger)
  rm(model_dnn)
  rm(RMSE_train)
  rm(RMSE_eval)
  rm(y_pred_train)
  rm(y_pred_val)
  rm(y_pred_test)
  rm(err_tr)
  rm(err_v)
  rm(err_tst)

}

# save the RMSE values in a table
RMSE_true_table <- data.frame("dropout_prob"= dropout,
                              "train_RMSE"= tr_err_true,
                              "validation_RMSE" = v_err_true)
print(RMSE_true_table)


# plot the RMSE over the epochs
pdf('plot_LC_drop_mix.pdf', width = 8, height = 6)
par(mfrow=c(1,1))
plot(RMSE_list[[6]]$train,type = "l", col = "red", ylim = c(0,1.3),
     xlab ="Epoch", ylab ="RMSE")
lines(RMSE_list[[6]]$eval,type = "l", col = "blue")
title(main ="Learning curve")
legend("topright", c("training RMSE","validation RMSE"),
       cex=0.8, col = c("red","blue"), pch = c(19,18), lty=1:1)
dev.off()


# plot the actual against predicted and residuals of the best model
pdf('act_pred_drop_mix.pdf', width = 12, height = 6)
par(mfrow=c(1,2))
plot(data_train.y, tr_pred_list[[6]],
     xlab = "actual", ylab = "predicted", xlim = c(-2.5,10), ylim =
c(-2.5,11),
     main = "Plot of actual vs predicted train values")
abline(a=0,b=1)
plot(data_test.y, tst_pred_list[[6]],
     xlab = "actual", ylab = "predicted",xlim = c(-2.5,10), ylim = c(-
2.5,11),
     main = "Plot of actual vs predicted test values")
abline(a=0,b=1)
dev.off()
```

```
pdf('resid_pred_drop_mix.pdf', width = 12, height = 6)
par(mfrow=c(1,2))
plot(tr_pred_list[[6]], data_train.y-tr_pred_list[[6]],
     ylab = "Residuals", xlab = "Predicted values",
     main = "Plot of the train residuals")
abline(0, 0)
plot(tst_pred_list[[6]], data_test.y-tst_pred_list[[6]],
     ylab = "Residuals", xlab = "Predicted values",
     main = "Plot of the test residuals")
abline(0, 0)
dev.off()
```