UCLouvain

FACULTE DES SCIENCES

Faculté des sciences

Autoregressive Time Series Forecasting with Neural Networks

Author: **Gabriel BAILLY**
Supervisor: **Rainer VON SACHS**
Reader: **Christian HAFNER**
Academic year 2021–2022
Master [120] en science des données, orientation statistique

Acknowledgments

First of all, I would like to thank my family for their continuous support during my studies: without you, this work would not have been possible. Thanks a lot to Naim Haddi without whom I would not have taken the path of statistics. And of course thank you to all my other beloved ones who supported me.

Next, I would like to thank Johannes Lederer for answering a few questions and giving some really helpful advices; Robert Nowak who gently shared some of his own experiments; and also to Pr. Christian Hafner who accepted to read the thesis.

Finally, last but not least, I would like to thank Pr. Rainer von Sachs who supervised this work and was always very kind and patient! The process was always very natural and our meetings really helped a lot to find which interesting path this work could take. Vielen Dank für Ihren zeitliche Aufwand!

*

This dissertation has been prepared in partial fulfillment of the requirements for the master degree in data science (statistics orientation) delivered by the Université Catholique de Louvain (Louvain-la-Neuve, Belgium).

Table of Contents

List of Figures

List of Tables

Notations

$\|\cdot\|_p$ . . . . . p-norm

$|a|$ . . . . . Absolute value, if a is a real scalar

$|S|$ . . . . . Cardinality of a set, if S is a set

$\otimes$ . . . . . Hadamard (or element-wise) product

$(\cdot)_+$ . . . . . ReLU (or positive part) function

$\sigma(\cdot)$ . . . . . Logistic function

$\mathbb{I}(\cdot)$ . . . . . Indicator function

$\phi(\cdot)$ . . . . . Activation function (generic)

a . . . . . Scalar (italic lower-case)

$\boldsymbol{a}$ . . . . . Vector (bold and italic lower-case)

$\boldsymbol{a}^\top$ . . . . . Transposed vector

$\mathbf{A}$ . . . . . Matrix (bold upper-case)

$\boldsymbol{X}$ . . . . . Random vector (bold and italic upper-case)

δ . . . . . Threshold of neuron activity

ℓ . . . . . Generic index of network layer

λ . . . . . Tuner (or multiplier, tuning parameter) of the penalization term

$\mathcal{L}(\cdot)$ . . . . . Likelihood function

$\ell_n(\cdot)$ . . . . . Log-likelihood function

$L(\cdot)$ . . . . . Loss function

ℓ_p . . . . . Subspace of finite p-norm vectors

$o(\cdot)$ . . . . . "Small o" in Landau notation

Ω . . . . . Set of parameters of a neural network

$\mathcal{R}(\cdot)$ . . . . . Risk function

$\nabla\mathcal{R}(\Omega)$ . . . . . Gradient of the risk with respect to every parameters in Ω

σ^2 . . . . . Variance of a random variable

v_k . . . . . k-th outer-weight of a shallow neural network

ω_k (or $\boldsymbol{\omega}_k$) . . . . . k-th inner-weight (or k-th vector of inner-weights) of a shallow neural network

Abbreviations

AR	Autoregressive
ARMA	Autoregressive moving average
ARNN	Autoregressive neural network
CNN	Convolutional neural network
DGP	Data generating process
LSTM	Long short-term memory network
MSE	Mean squared error
NLAR	Nonlinear AR
NLARMA	Nonlinear ARMA
ReLU	Rectifier linear unit
RNN	Recurrent neural network

Chapter 1

Introduction

1.1 Time series forecasting

Time series analysis and prediction have been topics of interest since the beginning of statistical sciences in many domains. Many subjects like weather forecasting, analysis and prediction of economic growth, risk management, astronomy (with time series related to light measurement) strongly relate to methods developed in this subset of statistics. Nowadays, the increasing ability of systems to record and store large amounts of data allows time series topics to be even more important in a wide domain of industrial applications, such as demand forecasting or even more importantly energy load predictions.

The classical approach of statistical forecasting has been formalized by Box and Jenkins (1976), which is based on modelling linear relations between lagged observations of time-dependent processes, mainly referred to as ARIMA models. However, this approach does not work for numerous applications. In particular, using it to predict financial time series will fail, resulting in a new approach of modelling variance-covariance dependence between lagged observations with (G)ARCH models (Engle, 1982; Bollerslev, 1986). However, limitations of linear ARIMA modelling does not only concern financial time series: many processes might feature structural changes that lead to nonlinearity. In such contexts, the linear models also fail.

Consequently, extensions of nonlinear regression methods to time series data were developed in the 1980's. Among those, some parametric models (for example threshold autoregression, smooth transition autoregression, and exponential autoregressive models) might be useful for domain specific interpretation, while nonparametric methods (Nadaraya-Watson, local polynomials, nonlinear additive models and other smoothing estimators) offer very flexible models that may yield better forecasts and are useful if the problem is not related to interpretation. The interested reader may head to Hafner (1998) and Zivot and Wang (2006) for reviews of nonlinear time series models. Out of the large amount of such methods, neural networks – which came from the

machine learning and engineering literature – where investigated and evolved into advanced models such as RNNs (Elman, 1990) and LSTMs (Hochreiter and Schmidhuber, 1997), but where also studied later through ARNNs (Trapletti et al., 2000) as a nonlinear generalization of AR(p) models.

1.2 Methodology and contribution

The aim of this thesis is to compare ARNN(p) (i.e a shallow neural network model) with the standard linear AR(p), in the framework of univariate autoregressive time series. For this purpose, we use Monte-Carlo simulation procedures which will allow to quantify how this model and its architectures improve the forecasts over linear methods in the cases of three processes of interest.

Additionally, it will provide a few guidelines for model and architecture choices which are not trivial and might overcome some of the issues of naive "grid-searching" methods – i.e. testing the choice of hyper-parameters by comparing various combinations with cross-validation – which are generally too time-consuming for personal computers. In particular, we extend the results of Parhi and Nowak (2021) to autoregression: it shows the equivalence between ℓ_2 regularization and a particular ℓ_1 regularization on neurons coefficients. This equivalence shows that using classical ℓ_2 and classical ℓ_1 penalties on neural networks will both result in estimation of sparse networks, i.e. networks where a lot of parameters are approximately equivalent to zero and do not contribute to the output of the network, but ℓ_2 penalties leads to "neuron" sparsity over coefficients sparsity. This strategy allows to choose automatically the number of hidden neurons during the optimization.

1.3 Structure of the thesis

Statistical Time Series Analysis. Chapter 2 defines the concept of stationarity and reviews the classical framework of forecasting univariate stationary time series. The aim of this chapter is to provide linear and nonlinear models which can be used as a benchmark against neural networks, which are a subset of machine learning methods.

Neural Networks. Chapter 3 defines the framework of neural networks as a regression method. We first define the class of "shallow" networks as a general class of functions and explain the motivations behind using such networks to approximate nonlinear functions. Then, we review the estimation method as well as optimization strategies to estimate the parameters, some of the most commonly used activation functions, and we finally explain that neural networks with those activations are identified only up to a set of specific

transformations. The aim of this chapter is to provide the necessary theory to later omit these explanation when presenting the ARNN model.

Penalization Methods. Chapter 4 reviews ℓ_2 and ℓ_1 regularization for linear regression models, as well as their extensions to neural networks. The aim of ℓ_2 penalization is to obtain functions with coefficients of lower amplitude, to obtain more stable models. The aim of ℓ_1 penalization is to make "parameters" selection by shrinking them to zero. Additionally, we review the results of Parhi and Nowak (2021) which states the equivalence of the two regularization methods in the framework of neural networks with a specific activation function. Then, using both methods will lead to sparse networks (where a lot of coefficients are zero). The aim of this chapter is to understand how using regularization can be used to avoid selecting a precise number of neurons in the network, i.e. the number of parameters that will lead to a satisfying nonlinear approximation.

Autoregressive Neural Networks. Chapter 5 presents the general class of ARNNs which can be used to forecast univariate autoregressive processes. We review different architectures, stationarity conditions, and the forecasting procedure. The aim of this chapter is to explain the main model to be analysed, and thereby define the main hyperparameters to vary in order to study the performances of ARNNs.

Simulation Study. Chapter 6 presents some experimental results of autoregressive neural networks on simulated data generated by (1) a smooth nonlinear autoregressive function of order 1, (2) a smooth nonlinear autoregressive function of order 2 and (3) a linear autoregressive function of order 1. We benchmark each experiment against a linear AR(p) fit, to quantify improvements of ARNNs over linear methods. We also vary hyperparameters of the ARNN such as the penalization method during optimization (no penalty, ℓ_2 and ℓ_1), the activation functions, and the inclusion of shortcut connections. Finally, we compare the sparse properties of ℓ_2 and ℓ_1 penalties on a single simulated sample.

Chapter 2

Statistical Time Series Analysis

Before going into neural networks models for prediction, let us review the classical statistical methods for time series analysis. We will only focus on stationary time series processes, especially on autoregressive ones. We will first define stationarity and introduce the framework of the AR(p) model. We will then define a general framework for nonlinear autoregressive processes.

We review these in order to use AR(p) models as benchmarks for simulation studies in later parts of this work, to assess whether autoregressive neural networks perform well and to which extent. It also provides a condition on the nonlinearity, which ensures that we use stationary nonlinear processes.

2.1 Stationary Processes

In many situations, time series feature trend and/or seasonal components. In this thesis however, we only consider processes which do not include these components: such series are called stationary processes. A strong definition of this concept is given by strict stationarity:

Definition 2.1 (Strict Stationarity). *A stochastic process $\{X_t\}_{t\in\mathbb{Z}}$ is strict-sense stationary if*

$$F(X_{t+h_1}, X_{t+h_2}, \ldots, X_{t+h_n}) = F(X_{h_1}, X_{h_2}, \ldots, X_{h_n}) \quad \forall t, \forall (h_1, \ldots, h_n), \forall n$$

where F is the cumulative distribution function of the unconditional joint distribution of the process at the specified times.

However, such a definition is generally too restrictive. We will then refer to the concept of weak stationarity when discussing stationary time series in the next chapters:

Definition 2.2 (Weak Stationarity). *A stochastic process $\{X_t\}_{t\in\mathbb{Z}}$ is weakly stationary if the three following properties are simultaneously verified:*

1. *its expected value is constant:* $\mathbb{E}(X_t) = \mu < \infty$, $\forall\, t \in \mathbb{Z}$
2. *its variance is finite:* $\mathbb{E}(X_t^2) < \infty$, $\forall\, t \in \mathbb{Z}$
3. *the covariance has an invariant structure:*
   $\mathrm{Cov}(X_r, X_s) = \mathrm{Cov}(X_{r+t}, X_{s+t})$, $\forall\, r, s, t \in \mathbb{Z}$

We will define the concept of white noise, which is essential to explain later the general class of linear processes.

Definition 2.3 (White Noise). *A stochastic process* $\{\varepsilon_t\}_{t\in\mathbb{Z}}$ *is a white noise process if*

1. $\mathbb{E}(\varepsilon_t) = 0$, $\forall t \in \mathbb{Z}$
2. $\mathbb{E}(\varepsilon_t^2) = \sigma_\varepsilon^2$, $\forall t \in \mathbb{Z}$
3. $\mathrm{Cov}(\varepsilon_t, \varepsilon_s) = 0$, $\forall t \neq s$ *in the case of weak white noise, thereby denoted as* $\varepsilon_t \sim \mathrm{WN}(0, \sigma_\varepsilon^2)$
4. *The* ε_t *are independent and identically distributed in the case of strong white noise, thereby denoted as* $\varepsilon_t \sim \mathrm{IID}(0, \sigma_\varepsilon^2)$

In other words, a white noise process is random and therefore unpredictable. In the following parts of this document, the notation ε_t is therefore used to denote an error term generated by a $\mathrm{WN}(0, \sigma_\varepsilon^2)$ process at time t, often referred to as the "innovations" of a process. This process is used as the building block of linear processes, and is interpreted as the residuals of the statistical and neural models of prediction.

2.2 Linear Processes

2.2.1 General definition

The class of linear processes allows for building theoretical models, which consist of linear combinations of innovations.

Definition 2.4 (Linear Process). *A stochastic process* $\{X_t\}_{t\in\mathbb{Z}}$ *with mean* μ *is a (general) linear process if it has the representation*

$$X_t = \mu + \sum_{k=-\infty}^{\infty} b_k \varepsilon_{t-k} \,, \tag{2.1}$$

where $\{\varepsilon_t\}_{t\in\mathbb{Z}}$ *is a sequence of white noise with variance* σ_ε^2 *and where*

$$\sum_{k=-\infty}^{\infty} b_k^2 < \infty \,. \tag{2.2}$$

The condition defined in Equation (2.2), that the sequence of coefficients is square-summable, ensures that a truncated linear process converges in mean square to X_t, i.e. when k gets large, the coefficients $|b_k|$ rapidly decay to 0, i.e, the process is weakly dependent: the covariance between X_t and X_{t+s} decays rapidly fast and therefore X_t mostly depends of white noise which occurred at times close to t.

Proposition 2.1. *If $\{X_t\}$ is a linear process as defined in Definition 2.4, then it is weakly stationary with $\mathbb{E}(X_t) = \mu$ and $\mathrm{Cov}(X_t, X_{t+s}) = \sigma_\varepsilon^2 \sum_{k=0}^{\infty} b_k b_{k+s}$ for all t.*

The proof is given in Brockwell and Davis (1991).

2.2.2 AR(p) Process

Autoregressive processes of order p are a special case of linear processes, which takes the form

$$\begin{aligned} X_t &= \varepsilon_t + A(B)X_t \\ &= \varepsilon_t + \sum_{k=1}^{p} a_k \, X_{t-k} \, . \end{aligned}$$

where B is the backshift operator, i.e. $B^d X_t = X_{t-d}$, and $A(B)$ is therefore the polynomial of backshifted values of X_t. One problem that arises is that we cannot determine at first sight with this definition if this process is stationary. To workaround this problem, we can first study it with an AR(1) process, where $X_t = a \, X_{t-1} + \varepsilon_t$. We can rewrite this equation recursively as

$$X_t = a^t \, X_0 + \sum_{k=1}^{t} a^{k-1} \varepsilon_{t-k+1}.$$

We observe that $\lim_{t\to\infty} a^t X_0 \to 0$ if and only if $|a| < 1$, ensuring the convergence of the process that would otherwise explode. In this case, if the condition on the coefficient a is fulfilled, this process has a stationary solution which has the form of a general linear process (also referred to as an MA(∞) representation):

$$X_t = \sum_{k=0}^{\infty} a^k \varepsilon_{t-k}.$$

and where the non-stationary part vanishes for sufficiently large t. Therefore, AR(1) processes are linear processes as defined in Equation (2.1) with $b_k = a^k$. Following Proposition 2.1, we can therefore conclude that AR(1) are stationary with mean $\mathbb{E}(X_t) = 0$ and a covariance structure

$$\mathrm{Cov}(X_t, X_{t+s}) = \sigma_\varepsilon^2 \sum_{k=0}^{\infty} a^k a^{k+s}$$

$$= \sigma_\varepsilon^2 \sum_{k=0}^{\infty} a^{2k} a^s$$
$$= \sigma_\varepsilon^2 \frac{a^s}{1-a^2}$$

where the last equality follows from the formula of geometric series. As we impose that $|a| < 1$, we observe that the covariance between X_t and X_{t+s} vanishes as s increases.

In the case of higher order autoregression, we have to use the generating polynomial of the model. Proposition 2.2 states the condition on the roots that ensure stationarity: the closer these roots are to the forbidden region $\{z : |z| \leq 1\}$, the slower its autocorrelation will decay and the longer it will take the process to behave in a stationary manner from its origin. In the case of fitting an AR(p) to data, it might even indicate the presence of a trend in the series.

Proposition 2.2. *An* AR(p) *process is stationary if and only if the roots of its generating polynomial* $A(z)$ *verify* $z_1, \ldots, z_p \in \{z : |z| > 1\}$*, or equivalently*

$$A(z) = 1 - \sum_{k=1}^{p} a_1 z^k \neq 0 \quad \forall |z| < 1 \, .$$

2.2.3 Estimation of AR(p)

In this section, we review maximum likelihood estimation (MLE) of the AR(p) parameters based on a sample $\mathcal{X}_T = \{X_t\}_{t=1,\ldots,T}$. Other methods, like least-squares (LSE) or Yule-Walker (YWE) estimators, are not as efficient especially for small sample size T, i.e.

$$\mathbb{E}[(\boldsymbol{u}^\top \widehat{\boldsymbol{\theta}}_{\text{MLE}} - \boldsymbol{u}^\top \boldsymbol{\theta})^2] \leq \mathbb{E}[(\boldsymbol{u}^\top \widetilde{\boldsymbol{\theta}} - \boldsymbol{u}^\top \boldsymbol{\theta})^2] \quad \forall \boldsymbol{u} \in \mathbb{R}^p \, ,$$

where $\boldsymbol{\theta} \in \mathbb{R}^p$ denotes the true vector of parameters, $\widehat{\boldsymbol{\theta}}_{\text{MLE}}$ its MLE, and $\widetilde{\boldsymbol{\theta}}$ its LSE or YWE. The intuition between this definition of efficiency is that $\widetilde{\boldsymbol{\theta}}$ will have larger variance than $\widehat{\boldsymbol{\theta}}_{\text{MLE}}$, which motivates for using the later instead.

Under the working assumption that $\{X_t\}$ is Gaussian, the likelihood of the AR(p) parameters is defined by

$$\mathcal{L}(\boldsymbol{\Gamma}_T) = (2\pi)^{-T/2} [\det(\boldsymbol{\Gamma}_T)]^{-1/2} \exp\Big(-\frac{1}{2} \boldsymbol{X}_T^\top \boldsymbol{\Gamma}_T^{-1} \boldsymbol{X}_T \Big)$$

where $\boldsymbol{X}_T = (X_1, \ldots, X_T)^\top$ and $\boldsymbol{\Gamma}_T$ is its $T \times T$ theoretical covariance matrix which can be expressed as a function of the parameters $\boldsymbol{a} \in \mathbb{R}^p$ and σ_ε^2. We then rewrite $\mathcal{L}(\boldsymbol{\Gamma}_T) = \mathcal{L}(\boldsymbol{a}, \sigma_\varepsilon^2)$. This likelihood being nonlinear in the parameters, the maximization is solved numerically with some iterative optimization algorithm (e.g. Newton-Raphson), where the LSE or YWE might be used as

initialization. Note that even if $\{X_t\}$ is not gaussian, this method is used and referred as "pseudo-MLE".

2.2.4 Forecasting AR(p)

Assume that the AR(p) parameters have been estimated by MLE with some sample $\mathcal{X}_T$, and denoted $\widehat{\boldsymbol{a}} = (\widehat{a}_1, \ldots, \widehat{a}_p)^\top$ and $\widehat{\sigma}^2_\varepsilon$. A one-step ahead predictor of X_{T+1} can be expressed as

$$\widehat{X}_{T+1} = \sum_{k=1}^{p} \widehat{a}_k X_{T+1-k} \ ,$$

and more generally, a multi-step ahead predictor is expressed as

$$\widehat{X}_{T+h} = \sum_{k=1}^{p} \widehat{a}_k X_{T+h-k}$$

where the unknown observations X_{T+h-k} with $h-k>0$ can be replaced by $\widehat{X}_{T+h-k}$. When the forecasting horizon h gets larger, $\widehat{X}_{t+h}$ will tend to the mean of the process because of the restrictions on the parameters.

One may also quantify an interval in which the forecast will fall with high probability. We present a method constructed by large-sample approximations, which ignores parameters estimation uncertainty. In the case of one-step ahead prediction, the prediction error is defined for $T \to \infty$ (Brockwell and Davis, 1991, pp.184) by

$$\mathbb{E}[(X_{T+1} - \widehat{X}_{T+1})^2] = \sigma^2_\varepsilon \ ,$$

i.e. the main source of uncertainty of the forecast is the variability of the innovation. Under the assumption that $\{X_t\}$ is Gaussian, a one-step ahead $(1-\alpha)$-level prediction interval is then given by

$$\left[\widehat{X}_{T+1} - z_{1-\alpha/2}\widehat{\sigma}_\varepsilon \ ; \ \widehat{X}_{T+1} + z_{1-\alpha/2}\widehat{\sigma}_\varepsilon\right]$$

with z_a the a-th quantile of the normal distribution and $\widehat{\sigma}_\varepsilon$ is the estimator of σ_ε obtained by MLE. In the case of the h-step ahead prediction, the prediction error is approached for $T \to \infty$ (Brockwell and Davis, 1991, Theorem 5.5.1) by

$$\mathbb{E}[(X_{T+h} - \widehat{X}_{T+h})^2] \approx \sigma^2_\varepsilon \sum_{k=0}^{h-1} b_k^2 \ ,$$

where the coefficients b_k come from its MA(∞) representation. As $b_0 = 1$, the prediction error will therefore increase with h. If we denote $\sigma_\varepsilon(h) := \sqrt{\sigma^2_\varepsilon \sum_{k=0}^{h-1} b_k^2}$ and under the assumption that $\{X_t\}$ is Gaussian, a h-step ahead

$(1-\alpha)$-level prediction interval is given by

$$\left[\widehat{X}_{T+h} - z_{1-\alpha/2}\widehat{\sigma}_\varepsilon(h)\ ;\ \widehat{X}_{T+h} + z_{1-\alpha/2}\widehat{\sigma}_\varepsilon(h)\right]$$

where $\widehat{\sigma}_\varepsilon(h)$ is calculated based on the parameters obtained by MLE.

2.3 Nonlinear Processes

In this section, we define the framework of nonlinear stochastic processes $\{X_t\}_{t\in\mathbb{Z}}$ of interest in this thesis. A general nonlinear autoregressive process of order p, namely NLAR(p), is defined as a function of p lags and an additive innovation, i.e.

$$X_t = f(X_{t-1}, \ldots, X_{t-p}) + \varepsilon_t \tag{2.3}$$

with $f : \mathbb{R}^p \to \mathbb{R}$, and where $\{\varepsilon_t\}$ is a white noise process, without loss of generality. Additionally, in order to restrict $\{X_t\}$ to the class of stationary nonlinear processes, we have to impose a restriction on f.

Condition 2.1. *The function f of an NLAR(p) process is such that*

$$f(\boldsymbol{x}) = \sum_{k=1}^{p} a_k x_k + o(\|\boldsymbol{x}\|_2) \quad \text{as } \|\boldsymbol{x}\|_2 \to \infty$$

where $\|\boldsymbol{x}\|_2$ is the Euclidean norm of the vector $\boldsymbol{x}$. It means that, when this norm is large, f is dominated by a linear function (Masry and Tjøstheim, 1995; Lu, 1998), i.e. the nonlinear part of f is bounded when away from the origin and eventually disappears in favour of the linear part. Additionally, the linear part must satisfy the constraint on the coefficients enunciated in Proposition 2.2.

This condition is not very restrictive and ensures that the process is stationary.

2.4 Conclusion

In this chapter, we reviewed the concept of stationarity and linear forecasting models. We defined the general class of linear processes which is very convenient to state conditions of stationarity for AR(p). We then explained the maximum likelihood estimation method of the model, as well as the properties of its predictions. Finally, we defined a very general form of nonlinear AR(p) process and some restrictions on the nonlinear part to ensure that we study stationary NLAR(p).

Chapter 3

Neural Networks

This chapter reviews the main concepts relevant to neural networks literature Statistical lexicon and notation is used to avoid confusion introduced by the wide lexicon of machine learning literature, which has a lot of synonyms for the same concept. Additionally, it is applied to classical regression problems: we do not use time series notation to avoid loss of generality.

3.1 Shallow Networks

Before going to the details of shallow networks, let us explain the most basic ANN model which is called the *perceptron* and was elaborated by Rosenblatt (1958) for classification tasks. The perceptron is a linear combination of a features vector $\boldsymbol{X} \in \mathbb{R}^p$ which is squashed by a function $\phi(\cdot)$ to compute a prediction $\widehat{Y}$:

$$\widehat{Y} = \phi\big(\boldsymbol{\omega}^\top \boldsymbol{X}\big)$$

where $\boldsymbol{\omega} \in \mathbb{R}^p$ is called the weight vector, and the function $\phi(\cdot)$ is called the activation function. The basic Rosenblatt's perceptron used the Heavyside function as an activation, which is piecewise linear and non-continuous. Later networks typically used squashing (i.e. bounded) nonlinear functions as activation, such as the logistic or the hyperbolic tangent. They are also continuous, non-constant and monotone increasing. But most modern neural networks use the so-called ReLU (Rectified Linear Unit) function, which is continuous and piecewise linear, but has numerous advantages over classical bounded activations.

The perceptron is unfortunately limited to very restricted problems (Minsky and Papert, 1969), which motivated for more complex variants of the perceptron: the general class of multi-layer perceptrons. Originally elaborated in order to solve classification problems, we can use them in a regression framework due to the universal approximation theorem (see Section 3.2).

We will now explain what is a shallow network. It is a model where the inputs from a vector $\boldsymbol{X} \in \mathbb{R}^p$ are processed by K neurons on a single hidden

layer. Those K neurons are then summed and go through another activation function to deliver the output $\widehat{Y}$. A regression model for prediction of a variable of interest Y given p random variables as a vector $\boldsymbol{X} = (X_1, \ldots, X_p)^\top$, using a shallow network, has the form

$$Y = f_\Omega(\boldsymbol{X}) + \varepsilon$$

Definition 3.1 (Shallow networks). *Shallow networks are functions that consist of linear combinations of K perceptrons, i.e. the class of functions $f_\Omega : \mathbb{R}^p \to \mathbb{R}$ of the form*

$$f_\Omega(\boldsymbol{X}) = v_0 + \sum_{k=1}^{K} v_k \, \phi\big(\boldsymbol{\omega}_k^\top \boldsymbol{X} + b_k\big) \tag{3.1}$$

where $\boldsymbol{\omega}_k \in \mathbb{R}^p$ for $k = 1, \ldots, K$ are vectors of weights between the input variables and the hidden nodes, namely "inner weights"; v_k for $k = 1, \ldots, K$ are the coefficients between the latent variables from the hidden layer and the output, namely "outer weights"; and v_0, $b_k \in \mathbb{R}$ for $k = 1, \ldots, K$ are bias coefficients associated to the output and hidden neurons, respectively. We denote the set of parameters Ω.

Figure 3.1 is a diagram representation of Equation (3.1): each green node is a an input from the vector $\boldsymbol{X}$, each yellow node sends biases to the next layer, each blue node is a an activation of the input signal, i.e. $Z_k = \phi(\boldsymbol{\omega}_k^\top \boldsymbol{X} + b_k)$, and the red node is the weighted sum of the K activations in the hidden layer, $\widehat{Y} = v_0 + \sum_{k=1}^{K} v_k \, Z_k$.

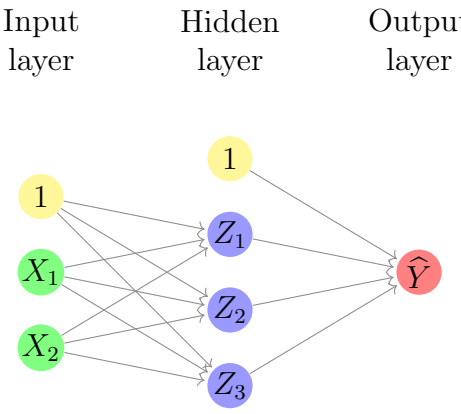


Figure 3.1: Shallow network with $p = 2$ and $K = 3$.

3.2 Universal Approximation Theorem

One of the key motivations for using multilayer perceptrons was given by the results of Cybenko (1989), Hornik (1991) and Leshno et al. (1993) which are

summarized in the following theorem:

Theorem 3.1 (Universal Approximation Theorem). *Let $\phi(\cdot)$ be a non-polynomial function, and let $\mathcal{C}(\mathbb{R}^p)$ be the set of continuous functions in $\mathbb{R}^p$. Then, shallow neural networks with $K < \infty$*

$$f_\Omega(\boldsymbol{x}) = v_0 + \sum_{k=1}^{K} v_k \phi\Big(\boldsymbol{\omega}_k^\top \boldsymbol{x} + b_k\Big)$$

are uniformly dense on compacta in $\mathcal{C}(\mathbb{R}^p)$, i.e. for every $f \subset \mathcal{C}(\mathbb{R}^p)$ there exist a shallow network such that

$$\sup |f_\Omega(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon \ ,$$

where $\epsilon > 0$ is an arbitrary small scalar.

In other words, this theorem states that there exists a single hidden-layer network that can approximate any measurable continuous function arbitrarily well (usually provided that the number K of hidden units is sufficiently large). The proof can be found in Leshno et al. (1993), and the interested reader can find wider discussions on this topic in Pinkus (1999).

3.3 Estimation of shallow networks

In the previous section, we explained the principles and motivations behind shallow networks. Now we will explain how to estimate the coefficients (or weights) of the model, which we denote by the set Ω for simplicity of the notation. In the framework of regression, one should estimate the set $\widehat{\Omega}$ such that the the network produces the most accurate predictions, based on a sample of size n, $(\boldsymbol{x}_i, y_i)_{i=1}^n$. The solution is given by the following minimization problem:

$$\widehat{\Omega} = \arg\min_{\Omega} \mathcal{R}(\Omega)$$

where $\mathcal{R}(\Omega)$ is the risk function, i.e. the expected value of the loss. In practice, one optimizes the empirical risk function:

$$\mathcal{R}_n(\Omega) = \frac{1}{n} \sum_{i=1}^{n} L(\Omega; y_i)$$

with $L(\Omega; y_i)$ a loss function. We omit the subscript n in the rest of this dissertation, as we always use the empirical risk to estimate neural networks.

To solve this minimization problem, we have to use variants of the gradient descent algorithm with error back-propagation. We first introduce the notion of *epoch* : it is defined as one pass of the optimization process over the whole dataset. In an epoch, there could be one iteration (*batch learning*) or several

(*stochastic learning* if as many iterations as observations in the sample, *mini-batch learning* if strictly between 1 and n iterations). In this section, we apply the techniques to mini-batch learning, where the number of observations in an iteration is denoted n^*. The idea is using the chain rule of derivation to estimate the weights. Let $\ell = 1, \dots, L$ be a layer number with L being the output layer, $\phi_j^{(\ell)}(\cdot)$ be an activation function at neuron j of the ℓ layer which produces an output $z_j^{(\ell)}$, and let $a_i^{(\ell+1)} = \boldsymbol{\omega}^{(\ell)\top} \boldsymbol{z}^{(\ell)}$ (see Figure 3.2). We recall that the gradient descent is the evaluation of $\frac{\partial \mathcal{R}(\Omega)}{\partial \omega_{ij}^{(\ell)}}$. We can decompose this derivative with respect to some weight (with $\omega_{ij}^{(L)} = v_j$):

$$\frac{\partial \mathcal{R}(\Omega)}{\partial \omega_{ij}^{(\ell)}} = \frac{\partial \mathcal{R}(\Omega)}{a_i^{(\ell)}} \frac{a_i^{(\ell)}}{\partial \omega_{ij}^{(\ell)}} = \delta_i^{(\ell)} z_j^{(\ell-1)}$$

where δ_i is just a lighter notation for the derivative of $\mathcal{R}(\Omega)$ with respect to a_i, and $z_j^{(\ell-1)} = X_j$ (the j-th input) if $\ell = 1$. We compute it as:

$$\delta_i^{(\ell)} = \begin{cases} \phi'(a_i^{(\ell)}) \dfrac{\partial \mathcal{R}(\Omega)}{\partial y_i} & \text{if output unit,} \\ \phi'(a_i^{(\ell)}) \displaystyle\sum_k \left(\delta_k^{(\ell+1)} \omega_{ki}^{(\ell+1)} \right) & \text{otherwise.} \end{cases}$$

We start the algorithm with some arbitrary weights and apply the network on a given input vector $\boldsymbol{x}_i$, which will produce an output $\widehat{y}_i$. Next we evaluate δ_i from the output unit and back-propagate it to evaluate the δ_i's from previous layers and use it to adjust the weights. This procedure repeats either until the maximum number of epochs is reached, or until convergence is reached (in which case the procedure stops, it is called early stopping).

Algorithm 3.1: BACK-PROPAGATION

```
initialize weights Ω_(0)
while not stop criterion do
    compute a_k^(ℓ) and z_k^(ℓ) for every k = 1, ..., K; ℓ = 1, ..., L
    for each (k, ℓ) do
        compute ∂R(Ω) / ∂ω_k^(ℓ)
        update ω_k^(ℓ) according to the specific optimization algorithm
```

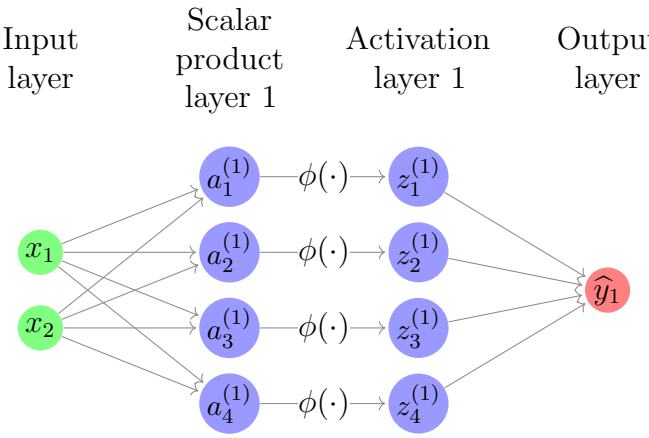


Figure 3.2: Graph representation of back-propagation

3.4 Optimization methods

We will explain five of the methods available in the Python package Keras (Chollet et al., 2015) : `SGD`, `AdaGrad` and `AdaDelta`, `RMSProp` and `Adam`. The latter is the most popular variant of stochastic gradient descent, because of its adaptive design. The term "training" will also be used to refer to the optimization process.

3.4.1 Stochastic Gradient Descent

The optimizer `SGD` stands for stochastic gradient descent. In this method, we compute the gradient on a subset of observations (in the framework of time series forecasting, the splitting is sequential). This subset, that is called batch in the optimizer's options, should in fact be called a mini-batch, as the batch is the whole dataset. However, for convenience and because the prefix *mini* is often unused, we will simply refer to these subsets as *batch*. Please note that the concept of stochastic optimization with batches will remain applicable for the next optimizers.

More formally, we have a dataset $\mathcal{D}$, we set $n_{\mathcal{B}}$ the size of batches, say 50 or 100. Then, we compute the gradient $\nabla\mathcal{R}(\Omega)$ as the derivative with respect to $\boldsymbol{\omega}$ in a batch $\mathcal{B}_k$, $k = 1, \ldots, n_{batch}$. For each batch, we update the weights with the following formula (with t being an index of batch iteration):

$$\Omega_{t+1} \leftarrow \Omega_t - \alpha\nabla\mathcal{R}(\Omega_t) + \beta\big[\Omega_t - \Omega_{t-1}\big]$$

The update rule has two hyper-parameters: first the *learning rate* α which is a small scalar (usually ≈ 0.01) in order to converge smoothly to a satisfying minimum. Then we have the *momentum* β (usually 0.9), which allows the previous update to contribute to the current one. This prevents oscillation of the updates and makes the convergence faster.

3.4.2 Adaptive Gradient Algorithms

In this class of optimizers, the algorithm is designed such that each weight has its own learning rate. When the weights have been significantly modified at previous iterations, we reduce their learning rate; and those which have only been slightly modified will have a bigger learning rate.

The first algorithm that we present in this section is called `AdaGrad`. The principle is the following: at each epoch, we compute the following matrix (where t is the current epoch iteration, and e is a generic index for epoch iterations):

$$\mathbf{V}_t = \sum_{e=1}^{t} \boldsymbol{g}_e \boldsymbol{g}_e^\top$$

where $\boldsymbol{g}_e$ is a lighter notation for the gradient of the risk at epoch e, i.e. $\nabla\mathcal{R}(\Omega_e)$. We take the diagonal of this matrix as a vector denoted $\boldsymbol{v}_t$ and then update the weights by the following :

$$\Omega_{t+1} \leftarrow \Omega_t - \alpha\, \boldsymbol{v}_t^{-1/2} \otimes \nabla\mathcal{R}(\Omega_t)$$

where $\otimes$ is the element-wise product, and the square root of the vector is also element-wise (which remains true for the rest of the section, when we will use the square root of a vector). The intuition behind the square root of the vector is given by the computation of $\mathbf{V}_t$: the update for each weight will depend on the euclidean norm of the previous derivatives from epoch 1 to t.

The second algorithm, `AdaDelta`, is a variant of AdaGrad where instead of using all past gradients for updating the learning rates, only a moving window of the past gradient is kept for weight updates. Then, the model can continue learning even after a lot of iterations: this variant is therefore *more robust* than the original one.

The `RMSProp` is the same idea as the previous algorithm : the learning rate is adapted for each weight. This time, we denote $g_{j,e}$ is the derivative of $\mathcal{R}(\Omega)$ w.r.t ω_j at epoch e. Then, let $\boldsymbol{v}_e$ be the vector with elements

$$v_{j,e} \leftarrow \beta\, v_{j,e-1} + (1-\beta) g_{j,e}^2 \qquad j = 1, \ldots, n_\omega$$

where β is a momentum parameter, which can be in this case interpreted as a forget rate of the previous epochs gradient. The update rule is the same as `AdaGrad`.

3.4.3 Adaptive Moment Estimation (Adam)

Kingma and Ba (2014) elaborated `Adam` as a variant of `RMSProp` which uses a supplementary moving average on the gradient during the update. Let us denote $\boldsymbol{m}_e$ the vector of exponential moving average on $\nabla\mathcal{R}(\Omega_t)$, we compute its elements by

$$m_{j,e} \leftarrow \beta_1\, m_{j,e-1} + (1-\beta_1) g_{j,e}$$

and we still compute $\boldsymbol{v}_e$ the same way as `RMSProp`, except we note its momentum parameter β_2. We then have to rescale these vectors in order to remove the bias from the first epochs (the initialization $\boldsymbol{v}_0$ and $\boldsymbol{m}_0$ is usually a vector of 0's, so it is biased towards 0):

$$\widetilde{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_2^t} \quad , \quad \widetilde{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t}$$

Finally, the update rule becomes :

$$\Omega_{t+1} \leftarrow \Omega_t - \alpha\Big(\widetilde{\boldsymbol{v}}_t^{1/2} + \epsilon\Big)^{-1} \otimes \widetilde{\boldsymbol{m}}_t$$

where $\epsilon > 0$ is a small scalar to prevent division by zero.

Finally, the `AMSGrad` algorithm (Reddi et al., 2019) is a variant of `Adam` that fixes a convergence issue: instead of using an exponential moving average of past squared gradients for parameters update, it uses the maximum of past squared gradients.

Algorithm 3.2: AMSGrad

initialize weights Ω_0
set α (typically between 0 and 0.1)
set $\beta_1, \beta_2 \in [0, 1)$ such that $\beta_1 < \sqrt{\beta_2}$ (in this thesis, fixed to $\beta_1 = 0.9$ and $\beta_2 = 0.999$)
set $\epsilon > 0$ arbitrary small
set $m_0 = 0$, $v_0 = 0$ and $\widehat{v}_0 = 0$
while *not stop criterion* **do**
    $\boldsymbol{g}_t = \nabla\mathcal{R}(\Omega_t)$
    $\boldsymbol{m}_t = \beta_1\boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$
    $\boldsymbol{v}_t = \beta_2\boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$
    $\widetilde{\boldsymbol{m}}_t = \boldsymbol{m}_t/(1-\beta_1^t)$
    $\widetilde{\boldsymbol{v}}_t = \boldsymbol{v}_t/(1-\beta_2^t)$
    $\widehat{\boldsymbol{v}}_t = \max(\widehat{\boldsymbol{v}}_{t-1}, \widetilde{\boldsymbol{v}}_t)$
    $\Omega_t = \Omega_{t-1} - \alpha(\widehat{\boldsymbol{v}}_t^{1/2} + \epsilon)^{-1} \otimes \widetilde{\boldsymbol{m}}_t$

3.5 Activation Functions

In this section, we review some of the most commonly used activation functions for regression problems: the logistic, the hyperbolic tangent and the ReLU functions. Nowadays, the first two functions are almost only used for regression problems, while the ReLU is widely used in both classification and regression tasks. Nonlinear activations are absolutely needed for a neural network to approximate a nonlinear function: linear activations would only lead to a linear neural network. Moreover, we recall that any function can be used as an activation as long as it is not a polynomial (Leshno et al., 1993): therefore, there exists infinitely many activation functions that may be used. The ones presented here are somehow preferred because of either their interpretation (logistic) or their mathematical properties (hyperbolic tangent and ReLU).

3.5.1 Logistic function

The logistic function is defined by

$$\begin{aligned} \sigma : \quad & \mathbb{R} \to (0,1) \\ & z \mapsto \frac{1}{1+e^{-z}} \end{aligned}$$

which is a smoothed version of the step function $\mathbb{I}(z \geq 0)$. It naturally arose as an activation due to its biological interpretation (as a "degree of neuron activation"). The logistic – on the contrary of the step function – is bounded, continuous and monotonically increasing, nonlinear. It belongs to the sigmoid family of activations, i.e. S-shaped functions "squashing" the real inputs to a bounded domain where $\lim_{z=-\infty} \phi(z) = 0$ and $\lim_{z=\infty} \phi(z) = 1$. Moreover, it is infinite-times derivable, which is a clear advantage over the step function for gradient-based optimization:

$$\frac{\partial}{\partial\, z}\sigma(z) = \sigma(z)\big(1-\sigma(z)\big)$$

3.5.2 Hyperbolic tangent function

The hyperbolic tangent function is defined by

$$\begin{aligned} \tanh : \quad & \mathbb{R} \to (-1,1) \\ & z \mapsto \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}\,. \end{aligned}$$

Moreover, one can show (Lederer, 2021) that we can express it as

$$\tanh(z) = 2\,\sigma(2z) - 1$$

and its derivative as

$$\frac{\partial}{\partial z}\tanh(z) = 1 - \big(\tanh(z)\big)^2 .$$

This function – as the logistic – is bounded, continuous, monotonically increasing, nonlinear, and infinite-times derivable. It has a similar shape as the logistic activation, but does not have quite the same motivation (which is, in the case of the logistic, a degree of activation from none to strong) because it covers a larger domain $(-1, 1)$. Nevertheless, it has an advantage over the logistic: the property of being symmetric around the origin, i.e. $\tanh(z) = -\tanh(-z)$. This property makes the network more likely to produce outputs centred around zero on average in the hidden layer (LeCun et al., 2012). Moreover, this results in larger first derivative: the first derivative of the logistic is bounded by 0.25, while that of tanh is bounded by 1.

3.5.3 Rectifier linear unit (ReLU) function

The rectifier linear unit or positive-part function, generally referred to as ReLU, is a piecewise linear and continuous function defined by

$$\begin{aligned}(\cdot)_+ : \quad & \mathbb{R} \to \mathbb{R}_+ \\ & z \mapsto \max\{0, z\} .\end{aligned}$$

This activation has been widely used as an effective activation for deep network models, requiring better optimization algorithms and computational speed to make these networks usable in practice. In particular, Jarrett et al. (2009) showed that ReLU activations enhance performances of convolutional neural networks (CNNs) for object recognition problems, further supported by the results of Nair and Hinton (2010) applied to restricted Boltzmann machines (RBMs).

This function increases the training speed due to the inexpensive nature of its computation and its derivative (which is although non-continuous at 0), i.e.

$$\frac{\partial}{\partial\, z}(z)_+ = \begin{cases} 1 & z \in (0, \infty) \\ 0 & z \in (-\infty, 0) \end{cases}$$

which makes the back-propagation easy to implement and faster than for more complex nonlinear functions. On the other side, because of its constantly zero part in the negative range, ReLU is subject to the "dying-ReLU" problem: if many inputs are negative, a large amount of neurons are "dead" or "inactive" (i.e. do not contribute to the output of the network) during the whole training, and the network will not be able to approximate complex functions and can even become a constant function (the network "dies"). However, such "dead" neurons are rarely inactive during the whole training process except in some

special cases (Lederer, 2021).

3.6 Identification of Neural Networks

One statistical issue with neural networks arise from identification of the parameters in such models. Indeed, Hwang and Ding (1997) showed that neural models are only identifiable up to a set of permutations between the parameters associated to the different hidden neurons. This unidentifiability of the neural network parameters Ω leads to unstable estimation, and often does not converge. To explain this result and state the theorem, we have to define the notions of reducibility (Sussmann, 1992) and redundancy of a neural network.

Definition 3.2 (Reducibility). *Let a neural network be of the form (3.1), with $\phi(\cdot)$ a non-constant, bounded and monotone increasing activation function. Then, the vector of parameters of the network Ω is called "reducible" if one of those holds:*

1. $v_k = 0$ *for some* $k = 1, \dots, K$.
2. $\boldsymbol{\omega}_i = \mathbf{0}$ *for some* $k = 1, \dots, K$ *and where* $\mathbf{0} = (0, \dots, 0)^\top \in \mathbb{R}^p$.
3. $(\boldsymbol{\omega}_i^\top, b_i) = \pm(\boldsymbol{\omega}_j^\top, b_j)$ *for some* $i \neq j$.

Hwang and Ding (1997) showed that a reducible neural network is also redundant, i.e. there exists another network with fewer hidden neurons which represents exactly the same function; and that under Condition 3.1 on the activation function, irreducibility of a network implies non-redundancy.

Condition 3.1 (Hwang and Ding (1997)). *The class of functions*

$$\{\phi(\boldsymbol{\omega}_k^\top \boldsymbol{X} + b_k)\} \cup \{1\} \ ,$$

for $k = 1, \dots, K$, is linearly independent for any positive integer K; and none of the polynomials

$$q_i(\boldsymbol{X}) = \boldsymbol{\omega}_i^\top \boldsymbol{X} + b_i \qquad i = 1, \dots, k$$

are sign equivalent, i.e. for $i \neq j$ we have $|q_i(\boldsymbol{X})| \neq |q_j(\boldsymbol{X})|$.

This condition is strongly related to the concept of multicollinearity: typically, in the case of linear regression, a model is unidentifiable if two columns are linearly dependent. In the present case of shallow networks, it means that the (shifted) functions generated by each neuron are linearly independent. This condition is therefore a generalization of multicollinearity to neural networks and gives a first insight of the identifiability framework for those models. The following definition states a specific family of transformations that leave the network unchanged.

Definition 3.3 (Invariant network transformations). *Let a neural network of the form (3.1) have parameters* v_0 *and* $\boldsymbol{\mu}_k^\top = (v_k, b_k, \boldsymbol{\omega}_k^\top)$ *for* $k = 1, \ldots, K$*. There are two kinds of transformations on* Ω*, which generate a family of* $2^K K!$ *elements, that leave (3.1) invariant:*

1. *The function is unchanged if we permute* $\boldsymbol{\mu}_i$ *and* $\boldsymbol{\mu}_j$*, for* $i \neq j$*.*
2. *If the network is redundant with, for some* $i \neq j$*,* $(\boldsymbol{\omega}_i^\top, b_i) = -(\boldsymbol{\omega}_j^\top, b_j)$*; the parameters* $(v_0, \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_i, \ldots, \boldsymbol{\mu}_k)$ *and* $(v_0 + v_i, \boldsymbol{\mu}_1, \ldots, -\boldsymbol{\mu}_i, \ldots, \boldsymbol{\mu}_k)$ *give the same value to (3.1) and hence the same distribution of the response variable.*

This family of transformation leaves the network output unchanged, i.e., for an input vector $\boldsymbol{x}$ and two sets of parameters Ω^* and Ω^{**} belonging to this family, $f_{\Omega^*}(\boldsymbol{x}) = f_{\Omega^{**}}(\boldsymbol{x})$. These transformations form the set of functions up to which the network is identifiable.

Theorem 3.2 (Hwang and Ding (1997)). *Let* $\phi(\cdot)$ *be an activation function satisfying Condition 3.1 and the vector of parameters* Ω *be irreducible. Then,* Ω *is identifiable up to the family of transformations enunciated in Definition 3.3.*

This theorem therefore defines a restricted framework of neural models parameters identification. It is sufficient that Condition 3.1 and irreducibility of the network (i.e. no useless neuron or pair of neurons) holds for the parameters to be identifiable up to a set of permutations between neurons. This condition is satisfied for the logistic and hyperbolic tangent functions (Trapletti et al., 2000) and for the ReLU function if $(\boldsymbol{\omega}_i, b_i)$ and $(\boldsymbol{\omega}_j, b_j)$ are linearly independent for $i \neq j$ (He et al., 2020, Theorem 2.1). This definition of identifiability provides some conditions on the network that, if considered when fitting neural networks, greatly improve convergence of the training procedure, thus leading to more stable estimation. Additionally, a regularization strategy during the optimization process might improve convergence to a consistent solution (i.e. where the values of the parameters follow a similar distribution between different trials of estimating the network).

3.7 Conclusion

In this chapter, we reviewed some essential aspects of neural network models. We defined the class of shallow networks, and stated that such functions are universal approximators of continuous functions. We then reviewed the estimation method and some advanced optimization algorithms to approach a solution. The different activation functions that will be compared in the simulation study were presented, and we finally discussed some issues about the parameters identification. In the next chapter, we will extend the estimation part by reviewing methods for regularizing the parameters during the optimization.

Chapter 4

Penalization Methods for Neural Networks

Choosing the best architecture for a neural model can be challenging. The architecture of a network can be seen as the set of hyperparameters of the network, such as the number K of hidden neurons, the number L of hidden layers, and the activation function. A widely used technique is grid-searching the optimal architecture, which is basically cross-validation over all combinations of parameters. However, this procedure is very time consuming especially if one wants to evaluate a wide range of parameters, especially the number K of hidden neurons (which might as well not be the same for every hidden layers). Hastie et al. (2009) recommend to choose rather too many hidden neurons than necessary to ensure enough flexibility to capture nonlinearities adequately; however, such choices may inevitably lead to overfitting the data. This motivates for using ℓ_2 and ℓ_1 penalization methods, i.e. adding a penalty to the loss function in order to either shrink the weights sufficiently for the network to generalize well, or to force some of the weights to be equal to zero and therefore remove the irrelevant neurons.

In this chapter, we review different regularization techniques widely used in classical linear models and their application to neural models. Furthermore, we will use the results of Parhi and Nowak (2021) to answer the choice of the activation function which is, in the case of penalization, related to the selection of the number K of hidden neurons.

4.1 ℓ_2 Regularization

4.1.1 Linear regression case

ℓ_2 regularization, also called Ridge regression in the framework of linear models, originates from a solution to solve ill-posed regression problems, see e.g. Hoerl and Kennard (1970). Typically, when p is very large or $p > n$, the classical

ordinary least squares (OLS) estimator $\widehat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\boldsymbol{y}$ is impossible to solve because the matrix $\mathbf{X}^\top\mathbf{X}$ is ill-conditioned or singular and therefore non-invertible. The identification problem arising from ill-conditioned matrices is linked to the phenomenon of highly correlated variables (imperfect collinearity of columns in the matrix $\mathbf{X}$) and will especially result in large standard errors of the estimated parameters, as well as issues for interpreting the coefficients (which could even have the opposite sign of the true parameter in the data generating process).

The idea of Ridge regression is to render the $\mathbf{X}^\top\mathbf{X}$ well-conditioned by adding a small term $\lambda \in \mathbb{R}^+$ on the diagonal. The estimator then takes the form $\widehat{\boldsymbol{\beta}}_{\text{Ridge}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_p)^{-1}\mathbf{X}^\top\boldsymbol{y}$, where $\mathbf{I}_p$ is the identity matrix of size $(p \times p)$. This estimator is the solution to

$$\widehat{\boldsymbol{\beta}}_{\text{Ridge}} = \arg\min_{\boldsymbol{\beta}} (\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^\top(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}) + \lambda\|\boldsymbol{\beta}\|_2^2 \,,$$

which is itself the Lagrangian of the following quadratic constrained optimization problem (with some $c \in \mathbb{R}^+$):

$$\widehat{\boldsymbol{\beta}}_{\text{Ridge}} = \arg\min_{\boldsymbol{\beta}} (\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^\top(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}) \quad \text{subject to} \quad \|\boldsymbol{\beta}\|_2^2 \leq c \,.$$

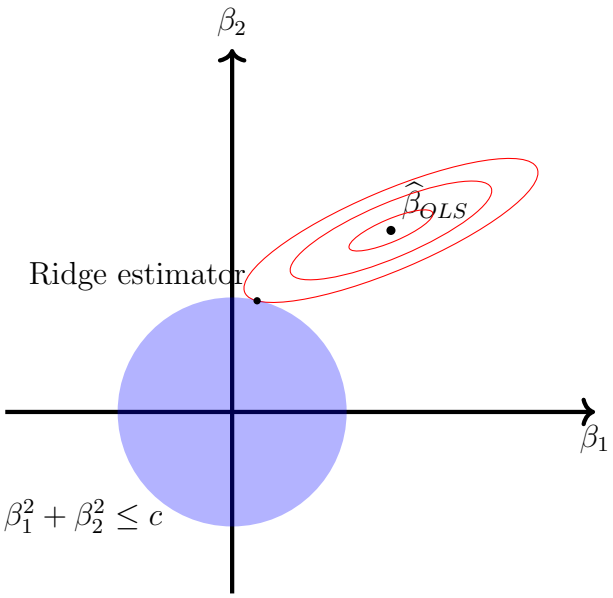


Figure 4.1: Geometric Interpretation of the ℓ_2 penalization

The multiplier λ is inversely proportional to the constraint c. We can show that $\text{Bias}[\widehat{\boldsymbol{\beta}}_{\text{Ridge}}]$ increases with λ, but $\text{Var}[\widehat{\boldsymbol{\beta}}_{\text{Ridge}}]$ decreases with λ; Ridge regression therefore needs the multiplier to be tuned such that there is a trade-off between those two moments, in order to minimize the prediction error. Additionally, geometrical interpretation of the Ridge regression solution can be

visualized in Figure 4.1. The blue region, a circle centred around the origin, is the constraint such that $\|\boldsymbol{\beta}\|_2^2 < c$; and the red ellipses are sets of points in the space of $\boldsymbol{\beta}$ which have the same sum of squared errors (SSE). The optimal Ridge solution to this constrained problem is the point where the ellipse touches the constrained region: we can easily see how it shrinks the parameters magnitude.

4.1.2 Neural network case

In the framework of neural networks, adding an ℓ_2-norm penalty also has a shrinkage interpretation. We define the ℓ_2-penalized risk for a shallow network as

$$\mathcal{R}^*(\Omega) = \mathcal{R}(\Omega) + \lambda \sum_{k=1}^{K} \left(v_k^2 + \|\boldsymbol{\omega}_k\|_2^2 \right) . \tag{4.1}$$

Using such method for neural models is widely referred to as "training neural networks with weight decay" in the machine learning literature. Using this notation, we can decompose the partial derivative as in Section 3.3 and show that using the penalty on weights sequentially on each layer ℓ, as done by the library Keras, is appropriate to apply ℓ_2 regularization:

$$\begin{aligned} \frac{\partial \mathcal{R}^*(\Omega)}{\partial \omega_{jk}^{(\ell)}} &= \frac{\partial \mathcal{R}(\Omega)}{\partial a_j^{(\ell)}} \frac{\partial a_j^{(\ell)}}{\partial \omega_{jk}^{(\ell)}} + 2\lambda \omega_{jk}^{(\ell)} \\ &= \delta_j^{(\ell)} z_k^{(\ell-1)} + 2\lambda \omega_{jk}^{(\ell)} \end{aligned}$$

Therefore the back-propagation will be of the same form as it was in Section 3.3, plus a term proportional to the weight.

4.2 ℓ_1 Regularization

4.2.1 Linear regression case

ℓ_1 regularization, also called LASSO regression (which is the acronym of *Least Absolute Shrinkage & Selection Operator*), was elaborated by Tibshirani (1996) as a variable selection method to determine the subset of predictors with the strongest impact on the response variable, without the drawbacks of subset selection techniques such as stepwise regression. This estimator is defined by (with some $c \in \mathbb{R}^+$):

$$\widehat{\boldsymbol{\beta}}_{\text{LASSO}} = \arg\min_{\boldsymbol{\beta}} (\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^\top (\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}) \quad \text{subject to} \quad \|\boldsymbol{\beta}\|_1 \leq c \tag{4.2}$$

where the Lagrangian is

$$\widehat{\boldsymbol{\beta}}_{\text{LASSO}} = \arg\min_{\boldsymbol{\beta}} (\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^\top (\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}) + \lambda \|\boldsymbol{\beta}\|_1.$$

Unless the covariates matrix $\mathbf{X}$ is orthonormal, there is no closed-form solution: it has to be solved numerically. There exists some algorithms to obtain solutions, such as Least Angle Regression (LARS) or Coordinate Descent. Furthermore, a geometrical interpretation of the minimization problem of Equation (4.2) can be found in Figure 4.2. The geometry of this problem implies that some coefficients will be forced to exactly zero as the constant c decreases, because of the hypercube nature of the constraint. This phenomenon makes the LASSO a "continuous subset selection" operator (Hastie et al., 2009). As for the ℓ_2 penalty, the multiplier λ is inversely proportional to the constraint c. The bias of the parameters increases with λ and their variance decreases with it: the multiplier should therefore be chosen such that the prediction error is minimized.

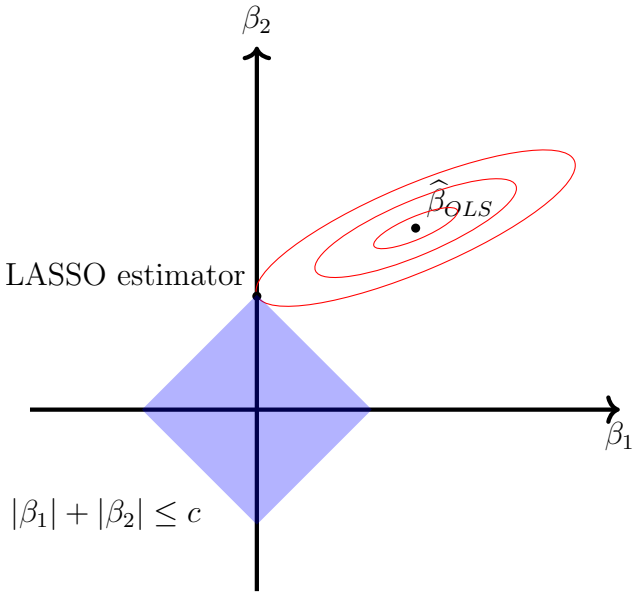


Figure 4.2: Geometric Interpretation of the ℓ_1 penalization

4.2.2 Neural network case

In the framework of neural networks, we define the ℓ_1-penalized risk for a network model as

$$\mathcal{R}^*(\Omega) = \mathcal{R}(\Omega) + \lambda \sum_{k=1}^{K} \left(|v_k| + |\omega_k| \right) . \tag{4.3}$$

Using this notation, we can decompose the partial derivative as before and show that using this penalty on each layers sequentially still allows to express

the derivative of the risk with respect to the weights as

$$\begin{aligned}\frac{\partial \mathcal{R}^*(\Omega)}{\partial \omega_{jk}^{(\ell)}} &= \frac{\partial \mathcal{R}(\Omega)}{\partial a_j^{(\ell)}} \frac{\partial a_j^{(\ell)}}{\partial \omega_{jk}^{(\ell)}} + \operatorname{sign}\big(\omega_{jk}^{(\ell)}\big)\lambda \\ &= \delta_j^{(\ell)} z_k^{(\ell-1)} + \operatorname{sign}\big(\omega_{jk}^{(\ell)}\big)\lambda\end{aligned}$$

4.3 Equivalence of ℓ_2 and ℓ_1 Regularization for ReLU Networks

In this section, we review the results of Parhi and Nowak (2021) which state that even when using ℓ_2-penalization, whose intuition is to shrink the size of the parameters to improve stability of the model, does in fact lead to sparse networks in a similar way as ℓ_1-penalization does (but needs empirically more epochs to reach the same level of sparsity). This equivalence is only true for when the activation function is ReLU, and the (implicit) ℓ_1 equivalence works a bit differently than classical "lasso-penalized" networks. These results are based on a simple algebraic result (Tibshirani, 2021): for any c,

$$\min_{ab=c}(a^2 + b^2) = 2|c| \tag{4.4}$$

which means that the arithmetic mean-geometric mean (AM-GM) inequality

$$\frac{1}{2}(a^2 + b^2) \geq ab$$

is tight when $a = b$ $(= \sqrt{c})$. We will exploit this inequality in the following section to explain the equivalence, which was first introduced in Neyshabur et al. (2014) as "implicit regularization".

4.3.1 Shallow Univariate case

Let $f_\Omega(\cdot)$ be a shallow, univariate ReLU network with K hidden-neurons; trained with a ℓ_2 penalty such that:

$$\mathcal{R}(\Omega) = \frac{1}{n}\sum_{i=1}^{n}\big(y_i - f_\Omega(x_i)\big)^2 + \lambda\sum_{k=1}^{K}\big(v_k^2 + \omega_k^2\big) \tag{4.5}$$

By the AM-GM inequality, we have (Neyshabur et al., 2014):

$$\frac{1}{2}\sum_{k=1}^{K}(v_k^2 + \omega_k^2) \geq \sum_{k=1}^{K}|v_k \cdot w_k|$$

Each term of the penalty is a sum of squares: therefore, we find $v_k^2+\omega_k^2 = 2|v_k\omega_k|$ at the global minimum. This allow us to rewrite (4.5) as an equivalent ℓ_1-regularized risk:

$$\mathcal{R}(\Omega) = \frac{1}{n}\sum_{i=1}^{n}\big(y_i - f_\Omega(x_i)\big)^2 + 2\lambda\sum_{k=1}^{K}|v_k\omega_k| \ .$$

This result shows that using an ℓ_2 penalty is implicitly equivalent to using ℓ_1 regularization on the product of inner and outer weights, resulting to neuron sparsity of the network. What we define as a "sparse" neural network is a network with only few active neurons. In the case of a shallow ReLU network, the concept of active neuron is summarized in the following definition .

Definition 4.1 (Active neuron). *The k-th hidden neuron of a shallow ReLU network is active if it contributes to the output, i.e.*

$$v_k(\boldsymbol{\omega}_k^\top x + b)_+ \neq 0 \ .$$

where $\boldsymbol{\omega}_k = \omega_k$ in the univariate case.

4.3.2 Shallow Multivariate case

Let $f_\Omega(\cdot)$ be a shallow, multivariate ReLU network with K hidden-neurons; trained with a ℓ_2 penalty such that:

$$\mathcal{R}(\Omega) = \frac{1}{n}\sum_{i=1}^{n}\big(y_i - f_\Omega(\boldsymbol{x}_i)\big)^2 + \lambda\sum_{k=1}^{K}(v_k^2 + \|\boldsymbol{\omega}_k\|_2^2) \tag{4.6}$$

$$v(\boldsymbol{\omega}^\top\boldsymbol{x} + b)_+ = v\|\boldsymbol{\omega}\|_2\frac{(\boldsymbol{\omega}^\top\boldsymbol{x} + b)_+}{\|\boldsymbol{\omega}\|_2}$$

We can, as in the univariate case, use the AM-GM inequality to show the implicit equivalence of ℓ_2 and a particular ℓ_1 penalty (Neyshabur et al., 2014):

$$\frac{1}{2}\sum_{k=1}^{K}(v_k^2 + \omega_k^2) \geq \sum_{k=1}^{K}|v_k|\cdot\|\boldsymbol{\omega}_k\|$$

Each term of the penalty is a sum of squares: therefore, we find $v_k^2 + \|\boldsymbol{\omega}_k\|_2^2 = 2|v_k|\|\boldsymbol{\omega}_k\|$ at the global minimum. This allow us to rewrite (4.6) as an equivalent ℓ_1-regularized risk:

$$\mathcal{R}(\Omega) = \frac{1}{n}\sum_{i=1}^{n}\big(y_i - f_\Omega(\boldsymbol{x}_i)\big)^2 + 2\lambda\sum_{k=1}^{K}|v_k|\|\boldsymbol{\omega}_k\|_2 \ ,$$

which has a sparse interpretation related to group lasso (Yuan and Lin, 2006),

and would be considered as such if the outer weights were fixed a priori, while it is optimized simultaneously with the inner-weights in this case. It can even be seen as an "adaptive group lasso" with K groups consisting of the parameters of each neuron, and where the "group weights" are adaptively chosen to minimize the risk. Parhi and Nowak (2021) define the penalization term $\sum_{k=1}^{K} |v_k| \|\boldsymbol{\omega}_k\|_2$ as the path-norm, i.e. the semi-norm of the network within a specific Banach space generated by sparse ReLU networks (the interested reader may head to the presentation of Nowak (2021) for a comprehensive introduction to the matter, and might also read the paper for deeper insights of this very technical subject).

4.3.3 Empirical study

We will now compare empirically the theoretical results reviewed in this section to regular ℓ_1 penalty; that is, we compare the sparsity of a network with the ℓ_2-penalized risk function described in Equation 4.1, and a network with the ℓ_1-penalized risk function described in Equation 4.3. We restrict to this comparison and do not compare the ℓ_2 and its implicit ℓ_1 equivalent (on the product of neuron weights), which was already empirically reviewed by Nowak (2021). The main objective is then not to validate that ℓ_2-penalization yields neuron-sparse networks, which is already empirically validated, but to which extent does it compare to the classical ℓ_1 or "network lasso". Moreover, most libraries directly implement such regularization and we should therefore study which of these already proposed methods will work better to obtain sparse results, especially when ℓ_1-penalty is widely used as "the" solution to obtain sparse networks.

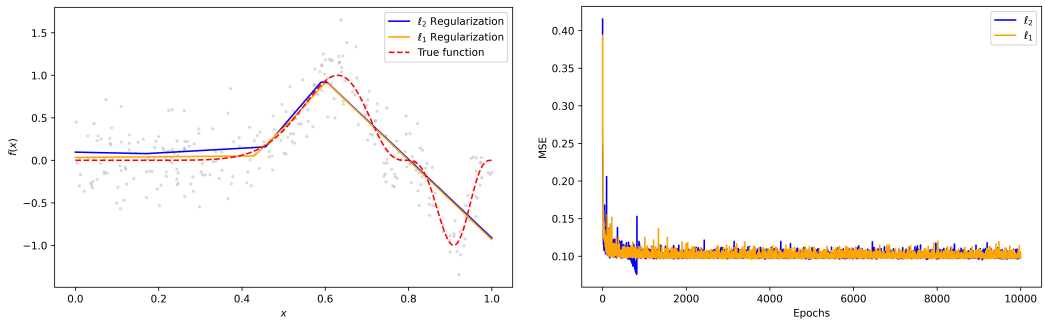
Before getting to comparisons, we first need to address a limitation of using numerical methods for estimation purpose. The coefficients will hardly be close to real zeros and therefore will not be truly sparse. Instead of waiting for the weights to reach real zeros, we have to choose some threshold δ under which we consider the trained weight to be zero. In the following experiment, we investigate how sparsity of the network evolves during training (how the number of active neurons decreases) for different δ. Please note the nomenclature used in this section: we refer to the multiplier of the penalization term, λ, as the *tuner* or *tuning parameter*; while we refer to δ, the value below which we consider a coefficient to zero, as the *threshold*.

We choose to study this phenomenon on simulated i.i.d. samples $(x_i, y_i)_{i=1}^{n}$ of size $n = 300$ generated by the data generating process (DGP) $y_i = m(x_i) + \varepsilon_i$, with

$$m(x) = \sin^3(2\pi x^3) \,, \tag{4.7}$$

and where $X \sim U[0, 1]$ and $\varepsilon \sim \mathcal{N}(0, 0.25)$. We use two shallow ReLU networks, with ℓ_2 and ℓ_1 penalization where we set $K = n$ hidden neurons (which over-parametrizes the function) and $\lambda = 10^{-5}$. The networks are estimated with the AMSGrad algorithm, where we set the learning rate to $\alpha = 0.05$ and the

number of epochs to 10001. We visualize in Figure 4.3 the estimated functions and the evolution of the loss during the optimization process.



(a) (b)

Figure 4.3: (a) Network approximations of function (4.7) and (b) MSE during training.

We can observe that the deterministic part of the DGP is quite nicely approximated in the interval $[0, 0.6)$ in which the first derivative of m does not vary much. In the interval $[0.6, 1]$ however, the first derivative of m oscillates quite a lot and we see that the estimated functions do not seem to approximate these inflexion points (which indicates that the tuning parameter λ is too large, which is not a big issue here as we mainly study the sparse properties). Concerning the MSE, we can observe that although showing small oscillation, it is quite consistent during training for both penalization methods, and does not appear to differ when the number of epochs is sufficiently large.

We now study the sparse behaviour of those networks, depending on which threshold is used to define whether the neurons are active. We then slightly modify the theoretical definition of active neurons, i.e. Definition 4.1, to fit for the use of thresholds.

Definition 4.2 (δ-active neuron). *The k-th hidden neuron of a shallow univariate ReLU network is δ-active if the two following inequalities hold:*

$$v_k > \delta \quad , \quad \omega_k > \delta \, .$$

In the multivariate case, the inequality is such that

$$v_k > \delta \quad , \quad \|\boldsymbol{\omega}_k\|_2 > \delta \, .$$

Table 4.1 reports the number of the number of δ-active neurons among $K = 300$ at the end of the optimization process. We can observe that lowering the threshold δ will decrease sparsity of the network in both penalization cases, but this phenomenon is stronger for ℓ_1 penalization. For $\delta = 0.1$ and 0.01, ℓ_1-penalized networks are sparser than ℓ_2 ones. However, when lowering the

magnitude of the threshold, ℓ_1 networks seem to have quite a high number of δ-active neurons.

We investigate this phenomenon by reporting in Figure 4.4 the sparsity ratio between the networks, i.e. the number of δ-active neurons in the ℓ_1 network divided by the number of δ-active neurons in the ℓ_2 network, against a long sequence of thresholds from 1e−5 to 1e−1. The ratio is approximately in the interval $[2/3; 3/4]$ when $\delta \geq$ 1e−2. However, when it decreases, we clearly see that the ratio explodes and is higher than 20 for really small thresholds. This indicates that although ℓ_1-penalized networks seem sparser with high thresholds, this sparsity is not robust to the choice of δ: this might then indicate that numerical algorithms converge better to consistently sparse solutions for ℓ_2-penalization than for ℓ_1-penalization, which is a very interesting result.

	# active neurons	
δ	ℓ_2	ℓ_1
10^{-1}	8	6
10^{-2}	12	9
10^{-3}	12	61
10^{-4}	12	177

Table 4.1: Comparison of network sparsity between ℓ_2 and ℓ_1 regularization, for different thresholds.

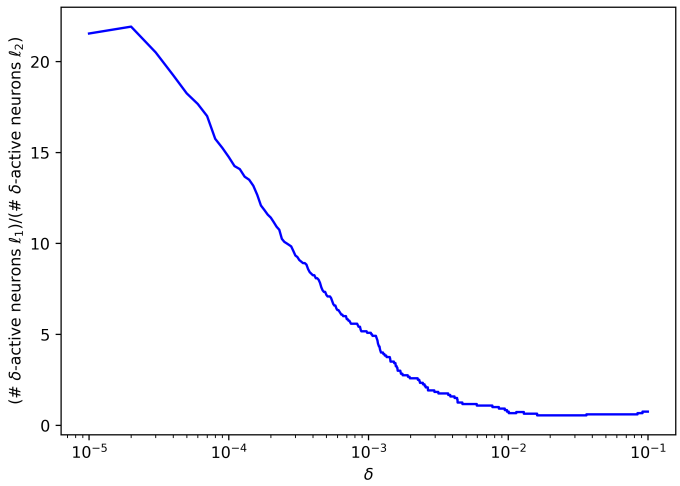


Figure 4.4: Sparsity ratio of ℓ_1 and ℓ_2 networks with different thresholds, represented in $\log_{10}$-scale for abscissa.

In other words, even if one would go for a ℓ_1-penalized network because of its fast convergence to a sparse solution, we recommend to use ℓ_2-penalization instead. This advice is motivated by the fact that, for the classical ℓ_1 penalty available in most packages and especially in Keras, the number of active neurons

is more sensitive to the choice of "below which small threshold" you consider the network to be sparse; while it is much more robust with respect to this choice if we consider ℓ_2-regularization. This observation is important because of the primary issue we mentioned in the beginning of this section: it is very hard to obtain real zeros with current optimization methods. Ideally, only values below a very tiny threshold should be considered as real zeros: suppose we are interested in approximating a function that maps into a domain of very small numbers, e.g. $[-0.1, 0.1]$, choosing $\delta = 0.01$ would probably be too large. Finally, this is even theoretically more interesting, as the implicit ℓ_1 equivalent of the regular ℓ_2 penalty is equivalent to the path-norm of the network and therefore has a different geometry than regular ℓ_1-regularization: it promotes neuron sparsity instead of weights sparsity, which is the desired task in most problems (especially given that the objective of this chapter is to use penalization methods to avoid choosing K with grid-searching procedures).

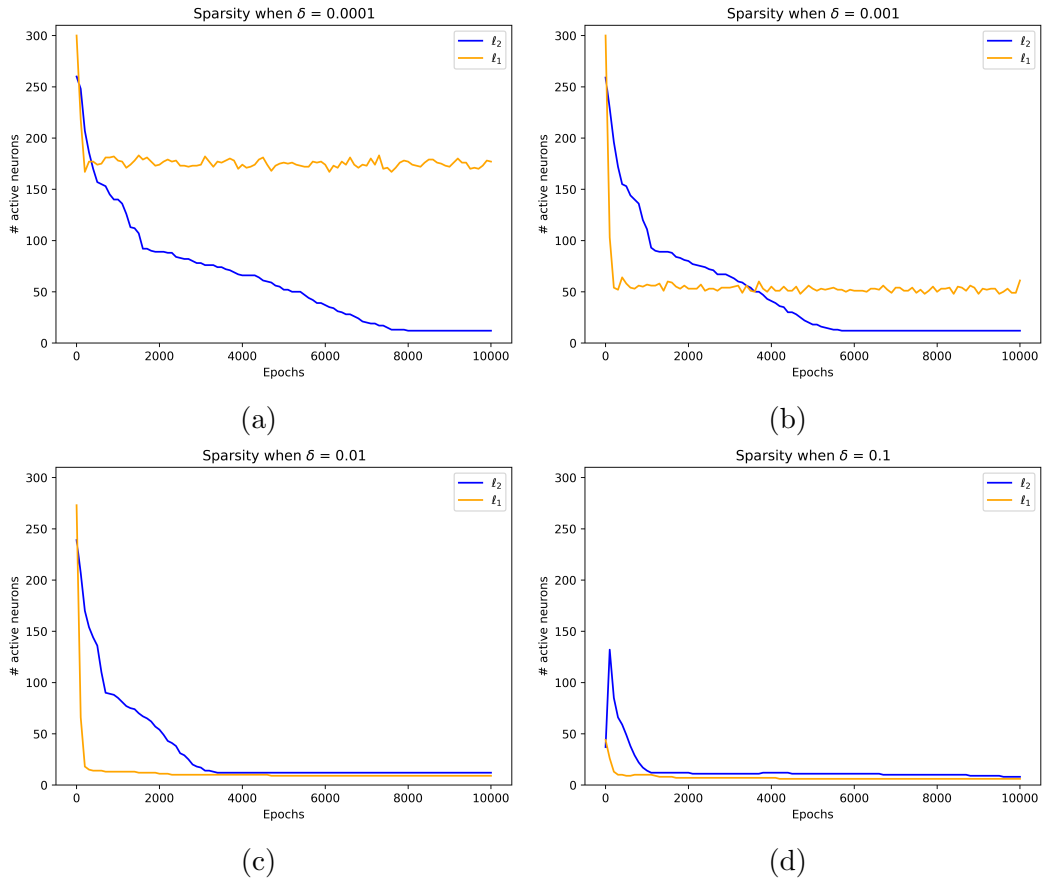


Figure 4.5: Sparsity of networks during training, for different thresholds.

We can also investigate how sparsity behaves for each regularizer during the optimization process. We first recall that in this experiment, the MSE is approximately constant for both networks after 1000 epochs. Once this fact has been established, we can observe the evolution of sparsity (by steps of

100 epochs) in Figure 4.5. In the case of the ℓ_2-penalized network, we observe for the different thresholds that the number of active neurons converges to a consistent number of neurons at the end of the training. This convergence to a consistently sparse result is faster as δ increases. In the case of the ℓ_1-penalized network however, we see that the number of active neurons converges very fast but is not consistent between different thresholds (this follows the observations from the previous paragraph). Moreover, for small δ, the sparsity will oscillate a lot throughout the optimization process, which isn't the case for ℓ_2-penalized networks.

Finally, we propose a pruning algorithm to redefine the trained sparse network into a model where only the δ-active neurons remain (which is important for computational purpose, as calculating many scalar products with tiny coefficients is unnecessary and time-consuming). We recall that δ-inactive neurons are either a zero map or a constant map (when $v_k, b_k > \delta$), in which case the neuron's contribution to the output is always $b_k \cdot v_k$. These facts motivate for a pruning procedure, detailed in Algorithm 4.1.

Algorithm 4.1: δ-ACTIVE PRUNING

Inputs: a trained, sparse shallow ReLU network f_Ω.
Define the set of neurons indices $\mathcal{I} = \{1, \dots, K\}$.
for $k = 1, \dots, K$ **do**
    if $|v_k| < \delta$ **then**
        remove k from $\mathcal{I}$
    else if $\|\boldsymbol{\omega}_k\|_2 < \delta$ **then**
        if $b_k \leq 0$ **then**
            remove k from $\mathcal{I}$
        else
            $v_0 \longleftarrow v_0 + b_k v_k$
Replace sparse network f_Ω by

$$f_\Omega(\boldsymbol{x}) = v_0 + \sum_{k \in \mathcal{I}} v_k (\boldsymbol{\omega}_k^\top \boldsymbol{x} + b_k)_+$$

4.4 Conclusion

In this chapter, we reviewed the Ridge and LASSO regularization methods for linear regression and their extension to shallow neural networks. We then reviewed the recent results of Parhi and Nowak (2021). They state that both ℓ_2 and ℓ_1 penalized neural networks will have a sparse set of parameters if optimized during a sufficiently long number of epochs, but that ℓ_2 penalty implicitly hunts for neuron sparsity (which has already been theoretically and empirically shown), while ℓ_1 results in sparsity of the weights. We then analyse the empirical properties of these results by comparing regular ℓ_2 and regular ℓ_1 regularizations (i.e. "ridge"-penalized and "lasso"-penalized networks). We define the concept of δ-active neurons – i.e. neurons where the weights have a magnitude lower than some small scalar δ – which is needed because current gradient descent algorithms will hardly attain the global minimum where the coefficients are real zeros. Using this definition to compare ℓ_2 and ℓ_1 with respect to different δ results in completely different levels of sparsity between the two penalization methods. Most importantly, these results show that even if ℓ_1 penalizations converges rapidly to a sparse solution (which makes it a better regularizer at first sight), ℓ_2 penalization converges to a solution where the sparsity is closer to a solution with real zeros. In other words, the experiment shows that ℓ_2 is more robust to the choice of a threshold than ℓ_1, and only needs sufficiently long training.

Chapter 5

Autoregressive Neural Networks

In this chapter, we will define the forecasting model of interest in this thesis: the ARNN(p). To simplify the notation of this section, let us denote $\boldsymbol{X}_t \in \mathbb{R}^p$ the vector of lags, i.e. $\boldsymbol{X}_t = (X_t, X_{t-1}, \ldots, X_{t-p+1})^\top$.

5.1 Definition of ARNN(p)

In Chapter 2, we defined the AR(p) process as a linear combination of a finite number of lags in the process $\{X_t\}_{t\in\mathbb{Z}}$. This model is useful and very convenient for interpretation purpose, but if the relation between X_t and the lagged observations of the process is nonlinear, AR(p) will fail to capture this relation and therefore to produce accurate forecasts. This motivates for nonlinear time series model. As we previously said in Chapter 3, neural networks models with a single hidden layer and an adequate activation function can approximate any function arbitrarily well. An nonlinear alternative to the classical autoregressive model is then given by the ARNN(p) neural model.

Definition 5.1. *A univariate autoregressive neural network of order p, denoted ARNN(p), is of the form*

$$X_t = f_\Omega\big(\boldsymbol{X}_{t-1}\big) + \varepsilon_t \tag{5.1}$$

where $\mathbb{E}(\varepsilon_t) = 0$ *,* $Var(\varepsilon_t) = \sigma_\varepsilon^2$ *and* $f_\Omega(\cdot)$ *is a neural network architecture. The set of parameters is denoted by* Ω*.*

5.1.1 Architectures

In this section, we review the two architectures that we will study for ARNNs. We will first define shallow ARNNs, and then we will define shallow ARNNs with shortcuts which are motivated by improving AR(p) predictions with the addition of a nonlinear part to the forecasting model.

Definition 5.2 (Shallow ARNN(p)). *A shallow univariate ARNN of order p consists of a shallow network where the inputs consist of lagged values of the target X_t, i.e.*

$$f_\Omega(\boldsymbol{X}_{t-1}) = v_0 + \sum_{k=1}^{K} v_k \phi\big(\boldsymbol{\omega}_k^\top \boldsymbol{X}_{t-1} + b_k\big) \ ,$$

where the set of parameters is equivalent from Ω in Definition 3.1.

This class of ARNNs is the most basic but is already powerful: as a shallow network, it benefits from the universal approximation property.

Definition 5.3 (Shallow ARNN(p) with shortcuts). *A shallow univariate ARNN of order p with shortcuts consists in the addition of a shallow ARNN(p) with a linear combination of the lagged values of the target X_t, i.e.*

$$f_\Omega(\boldsymbol{X}_{t-1}) = v_0 + \sum_{j=1}^{p} a_j X_{t-j} + \sum_{k=1}^{K} v_k \phi\big(\boldsymbol{\omega}_k^\top \boldsymbol{X}_{t-1} + b_k\big) \ ,$$

where the set of parameters is $\Omega \cup \{a_j\}_{j=1,\dots,p}$.

Shortcut connections are used a lot in the framework of convolutional neural networks (CNNs) which are used for completely different tasks, i.e. image recognition: in this context, it aims at mitigating the vanishing gradient problem which arises in deep neural network models (He et al., 2020). In the framework of the ARNN(p) model, its goal is to provide a linear part in the model that allows the hidden layer to model only the nonlinear part. It can therefore be viewed as a nonlinear improvement of a linear AR(p). Visualization of an ARNN with shortcuts is displayed in Figure 5.1.

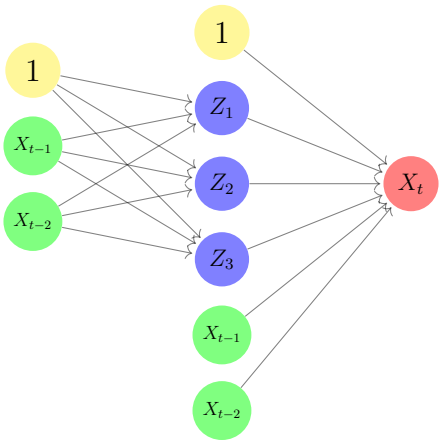


Figure 5.1: Shallow ARNN(2) with shortcuts and $K = 3$

5.1.2 Stationarity conditions

One question that naturally arises when using ARNNs to forecast autoregressive processes is to ensure whether these models are stationary. Leisch et al. (1999) showed that if $f_\Omega(\cdot)$ is an ARNN architecture without linear shortcuts, the ARNN(p) process is always (asymptotically) stationary, i.e. the solution of the (recursive) autoregressive equation will be stationary in the limit.

Theorem 5.1 (Leisch et al. (1999)). *Let $\{X_t\}$ be an ARNN(p) process as defined in 5.1. Furthermore, let $\mathbb{E}(|\varepsilon_t|) < \infty$ and the probability density function of ε_t be positive everywhere in $\mathbb{R}$. Then*

1. *If f_Ω is an ARNN without shortcuts and a bounded, continuous and monotonically increasing activation function, then $\{X_t\}$ is asymptotically stationary.*

2. *If f_Ω is an ARNN with shortcuts with a bounded, continuous and monotonically increasing activation function; and if the coefficients $\boldsymbol{a}$ of the linear part are such that its generating polynomial: $A(z) \neq 0$, $\forall z \in \mathbb{C} : |z| < 1$, then $\{X_t\}$ is asymptotically stationary.*

Therefore, a shallow univariate ARNN with an activation function that satisfies the condition of Theorem 5.1 item 1 ensures stationarity of the process. In the case where the ARNN has shortcuts, we need this assumption on the activation function as well as to impose the condition on the linear part exactly the same way as for linear AR(p). The generating polynomial $A(z)$ of the shortcuts shall have its unit roots in the region $\{z : |z| > 1\}$.

The issue with this Theorem is that it does not cover the ReLU activation function. To our knowledge, no result stating that shallow ReLU ARNNs are stationary exist, which is quite inconvenient if we want to extend the results from Section 4.3 to autoregression. However, we make the realistic assumption that if the nonlinear process $\{X_t\}$ satisfies Condition 2.1 and that the estimated ARNN with ReLU activation converges to a satisfying minimum, then the process generated by this network will be stationary.

5.2 Estimation of ARNN(p)

ARNNs such as the ones discussed in the previous section are estimated using error backpropagation, which is solved with numerical methods. The choice of the loss function $\mathcal{R}(\Omega)$ involved in the calibration of these naturally arises as the mean-squared error of prediction (MSEP), i.e. the MSE of one-step ahead predictions. This choice is motivated by the assumption that $\varepsilon_t \sim \mathcal{N}(0, \sigma_\varepsilon^2)$, so we have the probability density function of X_t conditionally on $\boldsymbol{X}_{t-1}$:

$$f_{X_t|\boldsymbol{X}_{t-1}}(z) = \frac{1}{\sigma_\varepsilon\sqrt{2\pi}} \exp\Big(-\frac{1}{2}\Big(\frac{X_t - \widehat{X}_t}{\sigma_\varepsilon}\Big)^2\Big)$$

Then the log-likelihood of X_t is given by

$$\ell_n(X_t|\Omega, \boldsymbol{X}_{t-1}) = -\frac{1}{2\sigma_\varepsilon^2}(X_t - \widehat{X}_t)^2 + c \,,$$

where $\widehat{X}_t = f_\Omega(\boldsymbol{X}_{t-1})$ and $c \in \mathbb{R}$ is a constant that can be ignored in the maximization. In this case, using the negative log-likelihood as a loss function is equivalent to minimize the MSEP:

$$\widehat{\Omega} = \arg\min_{\Omega} \frac{1}{T}\sum_{t=1}^{T}(X_t - \widehat{X}_t)^2$$

Once the network with $m = |\,\Omega\,|$ parameters is fitted to some sample $\mathcal{X}_T$ of size T, which can be a training set or an entire dataset (without loss of generality), we can estimate the variance of the residuals which can be used for constructing prediction intervals:

$$\widehat{\sigma}_\varepsilon = \sqrt{\frac{1}{T-m}\sum_{t=1}^{T}(X_t - \widehat{X}_t)^2}$$

5.3 Predictions of ARNN(p)

Assume that the ARNN(p) parameters have been estimated by minimization of the MSEP with some sample $\mathcal{X}_T$. The one-step ahead predictor of X_{T+1} can be expressed as

$$\widehat{X}_{T+1} = f_{\widehat{\Omega}}(\boldsymbol{X}_T) \,,$$

and more generally, the multi-step ahead predictor is expressed as

$$\widehat{X}_{T+h} = f_{\widehat{\Omega}}(\boldsymbol{X}_{T+h-1}) \,,$$

where the unknown observations X_{T+h-k} with $h - k > 0$ can be replaced by $\widehat{X}_{T+h-k}$.

As for linear predictors, one may need to quantify boundaries of values in which the actual X_{T+1} has a high probability to occur. Under the assumption that the process is conditionally gaussian, a $(1-\alpha)$-level prediction interval for the one-step ahead forecast is given by

$$\left[\widehat{X}_{T+1} - z_{1-\alpha/2}\widehat{\sigma}_\varepsilon \;;\; \widehat{X}_{T+1} + z_{1-\alpha/2}\widehat{\sigma}_\varepsilon\right]$$

where z_a is the a-th quantile of the normal distribution and $\widehat{\sigma}_\varepsilon$ corresponds to the residual standard deviation as defined in the previous section. For h-step ahead prediction however, such assumptions on the conditional distribution of the process do not work because of the unobserved X_{t-s}, $T < t - s < T + h$.

5.4 Conclusion

In this chapter, we defined the class of autoregressive neural networks and the architectures of interest in this thesis. We reviewed conditions on these functions, which ensure that the forecasts will behave in a stationary manner. Then, we motivated the use of the mean-squared error of prediction as the loss function and presented one-step ahead forecasting and the construction of its prediction intervals.

Chapter 6

Simulation Study

In this chapter, nonlinear time series are simulated to study the empirical behaviour of ARNNs, using Monte-Carlo procedures. We will study the impact of the *activation functions*, the *penalization* and the *shortcut connections*. Additionally, we will study how regularization behaves for neuron sparsity of ARNNs on a single simulated process.

6.1 Experimental Settings

Let us define the processes on which we will study performances.

Bump process. We use it for nonlinear autoregression of order 1:

$$X_t = -0.5X_{t-1} + \exp(-16X_{t-1}^2) + \varepsilon_t \tag{6.1}$$

where the second term is a rapidly decaying nonlinearity, which consists of a bump bounded by a linear part. This function can be visualized in Figure 6.1.

Saddle process. We use it for nonlinear autoregression of order 2:

$$X_t = -0.4\frac{(2 - X_{t-1}^2)}{(1 + X_{t-1}^2)} + 0.3\frac{(2 - (X_{t-2} - 0.5)^3)}{(1 + (X_{t-2} - 0.5)^4)} + \varepsilon_t \tag{6.2}$$

which allows to study the performance of an ARNN(2).

AR(1). We use it for linear autoregression of order 1:

$$X_t = -0.6X_{t-1} + \varepsilon_t \tag{6.3}$$

which allows to study if ARNNs fitting can perform well on linear processes and if using linear shortcuts increases the performance for such linear functions.

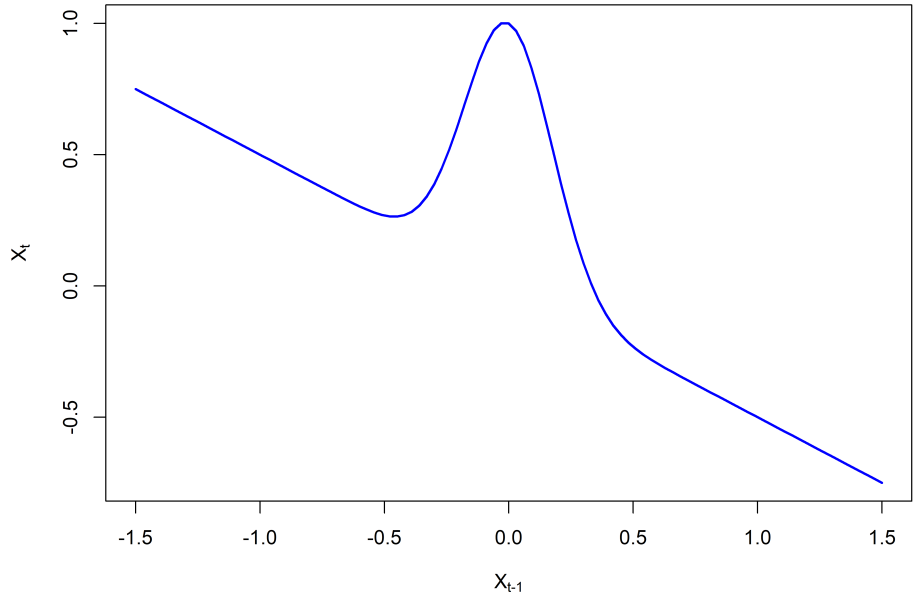
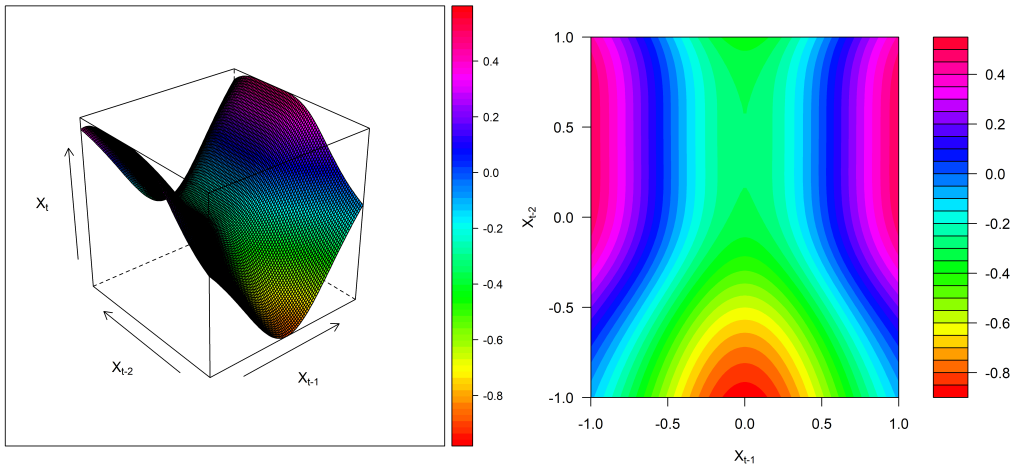


Figure 6.1: The Bump function defined in Equation (6.1)



(a) (b)

Figure 6.2: The Saddle function defined in Equation (6.2) in a perspective view (a) and in a contour view (b)

The two nonlinear processes satisfy Condition 2.1: they are dominated by a linear function as the inputs stray away from the origin.

The experimental procedure is a Monte-Carlo algorithm. When comparing the performances of different models on a particular process, we run the procedure M times where for each iteration, we simulate a sample $\{X_t\}_{t=1}^{T}$ following the process of interest. We fix a few parameters for every experiment:

- Number of simulations: $M = 50$
- Sample size: $T = 150$
- Burn-in period: 200
- Distribution of the noise: $\varepsilon \sim \mathcal{N}(0, 0.25)$
- Optimization algorithm ARNN(p): `AMSGrad`
- Estimation AR(p): MLE (with LSE as starting values).

The sample is generated after 200 burn-in realisations to ensure that the process is stationary. We fit each model to the first 80% of the sample, and use the remaining 20% (to simplify further notations, the size of this subset of the sample is T^*) for one-step ahead prediction. These predictions are then used to quantify the performance of the model at iteration m, by calculating the MSEP, i.e. the MSE of one-step ahead predictions:

$$\text{MSEP}^{(m)}(\Omega) = \frac{1}{T^*} \sum_{t=1}^{T^*} (X_t - \widehat{X}_t)^2$$

where $\widehat{X}_t = f_\Omega(\boldsymbol{X}_{t-1})$, following the same notation as Chapter 5. The final performance of the model is the mean of the M calculated MSEPs, which is the quantity reported in the tables of this chapter.

6.2 Activation functions

In this section, we study the performances of non-regularized ARNNs with the three activation functions described in Section 3.5, and different number of hidden neurons K. The invariant hyperparameters of the experiments are the following:

- Epochs: 500
- Batch size: 50
- Learning rate: 0.01
- Architecture: shallow ARNN(p) without shortcuts

We choose 500 epochs with batch size of 50, which is a fairly reasonable number of iterations to ensure convergence, associated with the fairly small learning rate (which is anyway adaptive to the gradient thanks to the `AMSGrad` algorithm).

Process (6.1). We compare the results of AR(1) fits with those of ARNN(1) on the Bump nonlinear process. The AR(1) is uniformly outperformed by ARNN(1), for any activation function, which is not surprising at all. Concerning the performance of the activation functions, the tanh activation stands out and outperform the logistic and ReLU. While the performances of the two bounded activations do not increase when $K > 3$, ReLU ARNNs need more hidden neurons to obtain similar forecasting accuracy.

		Activation		
K	AR(1)	σ	tanh	ReLU
2	0.504	0.314	0.281	0.372
3	0.506	0.303	0.273	0.335
4	0.512	0.299	0.262	0.301
5	0.504	0.292	0.265	0.315
6	0.511	0.297	0.270	0.303

Table 6.1: Performances (average MSEP of 50 simulations) of ARNN fit on process (6.1).

Process (6.2). We compare the results of AR(2) fits with those of ARNN(2) on the Saddle nonlinear process. The AR(2) is uniformly outperformed by ARNN(2) for every activations, and the gap between the linear and nonlinear fitting performances is very large (at least 20 times larger) which is not surprising if we consider that the DGP is highly nonlinear. A visual inspection still confirms that tanh activation works better than the other two, and increasing the number of hidden neurons K will also yield better forecast accuracy.

		Activation		
K	AR(2)	σ	tanh	ReLU
2	1.046	0.055	0.033	0.036
3	1.056	0.043	0.030	0.038
4	1.037	0.043	0.028	0.017
5	1.048	0.041	0.025	0.018
6	1.039	0.034	0.025	0.023

Table 6.2: Performances of ARNN fit on process (6.2).

Process (6.3). We compare the results of AR(1) fits with those of ARNN(1). In this case, the two forecasting models perform similarly, with AR(1) being a little more accurate than ARNN(1) – this is not surprising considering that the latter may slightly fit the noise. The three activation functions do not differ a lot, but the logistic surprisingly performs very slightly better than the other

activations (this could be confirmed by increasing the number of simulations, with enough computational power).

		Activation		
K	AR(1)	σ	tanh	ReLU
2	0.261	0.267	0.268	0.269
3	0.236	0.240	0.240	0.243
4	0.254	0.253	0.254	0.255
5	0.251	0.251	0.251	0.256
6	0.241	0.246	0.247	0.251

Table 6.3: Performances of ARNN fit on process (6.3).

6.3 Penalization

In this section, we study the performances of regularized ARNNs. We compare two regularizers, ℓ_2 and ℓ_1 as described in Chapter 4. We also vary the tuning parameter λ; and each time the tuning parameters is λ for ℓ_2 and 2λ for ℓ_1 (which is a fair compromise to obtain comparable performances). Finally, we vary the three activation functions that we already compared in the previous section. The invariant hyperparameters of the experiments are the following:

- Epochs: 2000
- Batch size: 50
- $K = 150$
- Learning rate: 0.05
- Architecture: shallow ARNN(p) without shortcuts

The number of epochs, batch size and learning rate are chosen in order to converge to a sparse solution – as we need longer training to achieve this than for non-regularized networks – while maintaining viable computational time. We also choose $K = 150$, i.e. the number of simulated observations, to over-parametrize the model and be sure that we have sparsity with respect to the amount of coefficients, which corresponds to $|\Omega| = (p+1)K + K + 1$.

Process (6.1). We compare the results of penalized ARNN(1) on the Bump nonlinear process. We recall that the AR(1) mean performance on this process was approximately 0.507. In the case of ℓ_2 penalization, it is uniformly outperformed by ARNN(1). The latter performs similarly for all three activations, except for the logistic whose performance degrades when λ increases. In the case of ℓ_1 penalization, AR(1) is still outperformed by ARNN(1) for the three

activations, but this regularizer does not seem to degrade the logistic performance when λ increases. Also, note that these two methods also outperform the non-penalized ARNNs, which can be explained by the fact that the number of active neurons after training is greater than the number of neurons selected in previous sections.

		Activation		
Reg.	λ	σ	tanh	ReLU
ℓ_2	10^{-6}	0.257	0.257	0.263
	10^{-5}	0.258	0.259	0.263
	10^{-4}	0.256	0.248	0.254
	10^{-3}	0.309	0.266	0.266
ℓ_1	$2 \cdot 10^{-6}$	0.267	0.264	0.270
	$2 \cdot 10^{-5}$	0.254	0.251	0.261
	$2 \cdot 10^{-4}$	0.261	0.259	0.267
	$2 \cdot 10^{-3}$	0.257	0.252	0.274

Table 6.4: Performances of penalized ARNN fit on process (6.1).

Process (6.2). We compare the results of penalized ARNN(2) on the Saddle nonlinear process. We recall that the AR(2) mean performance on this process was approximately 1.045. In the case of ℓ_2 penalization, it is uniformly outperformed by ARNN(2). The latter performs similarly for all three activations and tuning parameters, no specific phenomenon is detected based on this set of simulations. In the case of ℓ_1 penalization, AR(2) is still outperformed by ARNN(2) for the three activations which perform similarly, but this time increasing λ seems to slightly improve the performance. Note that fitting the Saddle process with penalized ARNNs has worse performances than non-penalized ones (this phenomenon is opposed to what we observed for the Bump process).

Process (6.3). We compare the results of penalized ARNN(1) on the linear autoregressive process. We recall that AR(1) fits had a mean performance of approximately 0.249. In the case of ℓ_2 penalization, the linear AR fit is still better than ARNNs, but the latter has only slightly worse performances. The three activations yield similar forecasting accuracy, but increasing the tuning parameter will slightly degrade the performance – which is surprising in the case of the ReLU activation, because fitting a linear process should need very few activations, and we would expect that such a case would happen if we increase λ for penalized ReLU ARNN (motivated by the theory developed in Section 4.3). In the case of ℓ_1 penalization, the performance is still only slightly worse than linear AR fit but, unlike ℓ_2 regularization, increasing λ will enhance the forecasting accuracy (which eventually performs similarly on average for

		Activation		
Reg.	λ	σ	tanh	ReLU
ℓ_2	10^{-6}	0.289	0.271	0.291
	10^{-5}	0.271	0.271	0.290
	10^{-4}	0.257	0.269	0.274
	10^{-3}	0.285	0.272	0.267
ℓ_1	$2 \cdot 10^{-6}$	0.301	0.275	0.294
	$2 \cdot 10^{-5}$	0.291	0.277	0.297
	$2 \cdot 10^{-4}$	0.259	0.263	0.266
	$2 \cdot 10^{-3}$	0.271	0.270	0.262

Table 6.5: Performances of penalized ARNN fit on process (6.2).

$\lambda = 2 \cdot 10^{-3}$). This phenomenon is coherent with what is expected if we penalize a lot for fitting a linear autoregressive process.

		Activation		
Regularizer	λ	σ	tanh	ReLU
ℓ_2	10^{-6}	0.251	0.247	0.252
	10^{-5}	0.252	0.253	0.254
	10^{-4}	0.255	0.254	0.256
	10^{-3}	0.260	0.257	0.260
ℓ_1	$2 \cdot 10^{-6}$	0.261	0.261	0.263
	$2 \cdot 10^{-5}$	0.254	0.249	0.257
	$2 \cdot 10^{-4}$	0.255	0.255	0.259
	$2 \cdot 10^{-3}$	0.249	0.248	0.249

Table 6.6: Performances of penalized ARNN fit on process (6.3).

6.4 Shortcut connections

In this section, we study the performances of regularized ARNNs with shortcuts. We compare two regularizers – ℓ_2 and ℓ_1 – and vary the tuning parameter λ; for each simulation, the tuner is λ for ℓ_2 and 2λ for ℓ_1, and we vary the three activation functions. The invariant hyperparameters of the experiments are the following:

- Epochs: 2000
- Batch size: 50
- $K = 150$
- Learning rate: 0.05

- Architecture: shallow ARNN(p) with shortcuts

The choice of the hyperparameters has the same motivations as previous sections, except that this time the number of coefficients corresponds to $|\Omega| = (p+1)K + K + 1 + p$.

Process (6.1). We compare the results of penalized ARNN(1) with shortcuts on the Bump nonlinear process. We recall that the AR(1) mean performance on this process was approximately 0.507. In the case of ℓ_2 penalization, it is uniformly outperformed by ARNN(1). The latter performs similarly for all three activations, except for the logistic whose performance degrades when λ increases (note that increasing the tuner will also degrade the other activations, but only slightly). In the case of ℓ_1 penalization, we make the exact same observations. If we compare to the previous section where we used the same models but without shortcuts, we do not observe very different results, except for the performance of the logistic using ℓ_1 penalization (without shortcuts: no degradation of forecasting accuracy when λ increases, unlike the shortcuts architectures).

		Activation		
Reg.	λ	σ	tanh	ReLU
ℓ_2	10^{-6}	0.259	0.258	0.274
	10^{-5}	0.262	0.265	0.263
	10^{-4}	0.278	0.260	0.264
	10^{-3}	0.420	0.300	0.273
ℓ_1	$2 \cdot 10^{-6}$	0.264	0.265	0.273
	$2 \cdot 10^{-5}$	0.269	0.263	0.270
	$2 \cdot 10^{-4}$	0.283	0.263	0.266
	$2 \cdot 10^{-3}$	0.398	0.292	0.299

Table 6.7: Performances of penalized ARNN with shortcuts fit on process (6.1).

Process (6.2). We compare the results of penalized ARNN(2) with shortcuts on the Saddle nonlinear process. We recall that the AR(2) mean performance on this process was approximately 1.045. In the case of ℓ_2 penalization, it is uniformly outperformed by ARNN(2). The latter performs similarly for all three activations, except for the logistic which degrades when λ increases. In the case of ℓ_1 penalization, we observe the same phenomenon. If we compare to the previous section, we observe that using shortcuts does not enhance performance – which even degrades for when using the logistic activation with large penalization.

		Activation		
Reg.	λ	σ	tanh	ReLU
ℓ_2	10^{-6}	0.289	0.271	0.291
	10^{-5}	0.271	0.271	0.290
	10^{-4}	0.257	0.269	0.274
	10^{-3}	0.285	0.272	0.267
ℓ_1	$2 \cdot 10^{-6}$	0.276	0.266	0.295
	$2 \cdot 10^{-5}$	0.283	0.276	0.286
	$2 \cdot 10^{-4}$	0.277	0.275	0.282
	$2 \cdot 10^{-3}$	0.304	0.268	0.264

Table 6.8: Performances of penalized ARNN with shortcuts fit on process (6.2).

Process (6.3). We compare the results of penalized ARNN(1) with shortcuts on the linear autoregressive process. We recall that AR(1) fits had a mean performance of approximately 0.249. In the case of ℓ_2 penalization, the linear AR fit is still better than ARNNs, but the latter has only slightly worse performances. The three activations yield similar forecasting accuracy, and increasing the tuning parameter will very slightly enhance the performance. In the case of ℓ_1 penalization, we observe the same phenomenon. The results of these experiments are coherent with what is expected if we penalize a lot for fitting a linear autoregressive process: the process being linear, the shortcut will be sufficient to fit a good forecasting model, and penalizing the nonlinear part of the network will mitigate the unnecessary parameters which might fit the noise and therefore degrade the performance.

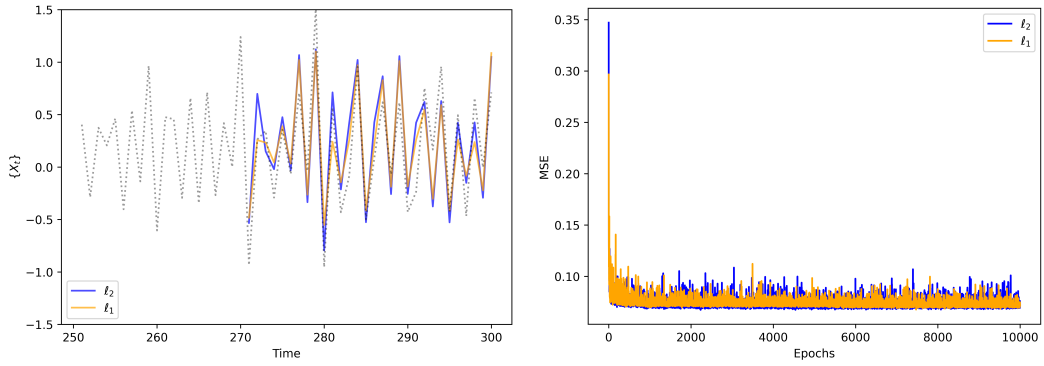
		Activation		
Reg.	λ	σ	tanh	ReLU
ℓ_2	10^{-6}	0.257	0.252	0.260
	10^{-5}	0.247	0.247	0.253
	10^{-4}	0.258	0.255	0.259
	10^{-3}	0.251	0.246	0.244
ℓ_1	$2 \cdot 10^{-6}$	0.258	0.252	0.258
	$2 \cdot 10^{-5}$	0.250	0.246	0.252
	$2 \cdot 10^{-4}$	0.261	0.258	0.262
	$2 \cdot 10^{-3}$	0.245	0.242	0.242

Table 6.9: Performances of penalized ARNN with shortcuts fit on process (6.3).

6.5 Sparsity

In this section, we study on a single experiment the sparse behaviour of penalized ARNN(p). We simulate $T = 300$ realisations of a NLAR process of order 1 generated from the DGP (6.1), to study whether the results of Section 4.3.3 hold for time-series data. We use two shallow ReLU ARNN(1) without shortcuts and set the number of hidden neurons to $K = T$, and train them with ℓ_2 and ℓ_1 regularization respectively. We set the number of epochs to 10001, the batch size to 50, and the learning rate to $\alpha = 0.05$.

We fit the two models to the simulated process, and visualize in Figure 6.3 that both networks make similar forecasts and have similar MSE during training – even if oscillations clearly occur, the loss is quite consistent during the optimization. We compute the sparsity ratio of those ARNNs (i.e. the number of δ-active neurons in the ℓ_1 network divided by the number of δ-active neurons in the ℓ_2 network), report the sequence of the ratio in Figure 6.4 and find out that it behaves in a similar fashion as the experiment in Section 4.3.3, but the explosion of the ratio starts below 10^{-3} instead of 10^{-2} previously. This phenomenon already indicates that our results of the previous experiment without time-dependence also applies to time-series data.



(a) (b)

Figure 6.3: (a) Forecasting process (6.1) with ARNNs, where the dotted gray line is the simulated process; (b) MSE during training.

We inspect the evolution of sparsity during training with respect to different thresholds δ in Figure 6.5, and we still observe that convergence of the ℓ_2-penalized network to a sparse solution is more robust than the ℓ_1-penalized one – and only needs sufficiently long training to attain sparsity even for small δ, while ℓ_1 is "stuck" to a much less sparse solution when we set the threshold sufficiently low. The message is then the same for time-series data than for classical i.i.d. observations, described in Section 4.3.3 : even if longer training is required, using ℓ_2-penalization leads to solutions closer to real sparsity (where the almost zero coefficients are very small floating numbers), which makes the choice of a threshold δ less important.

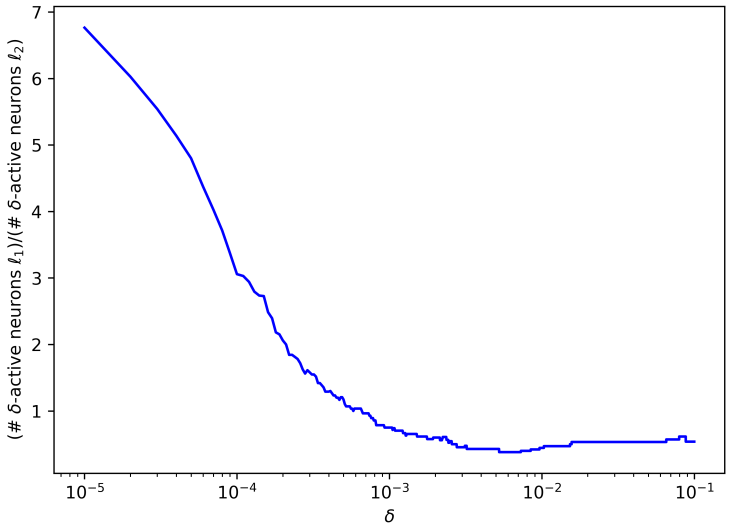


Figure 6.4: Sparsity ratio of ℓ_1 and ℓ_2 ARNNs with different thresholds, represented in $\log_{10}$-scale for abscissa.

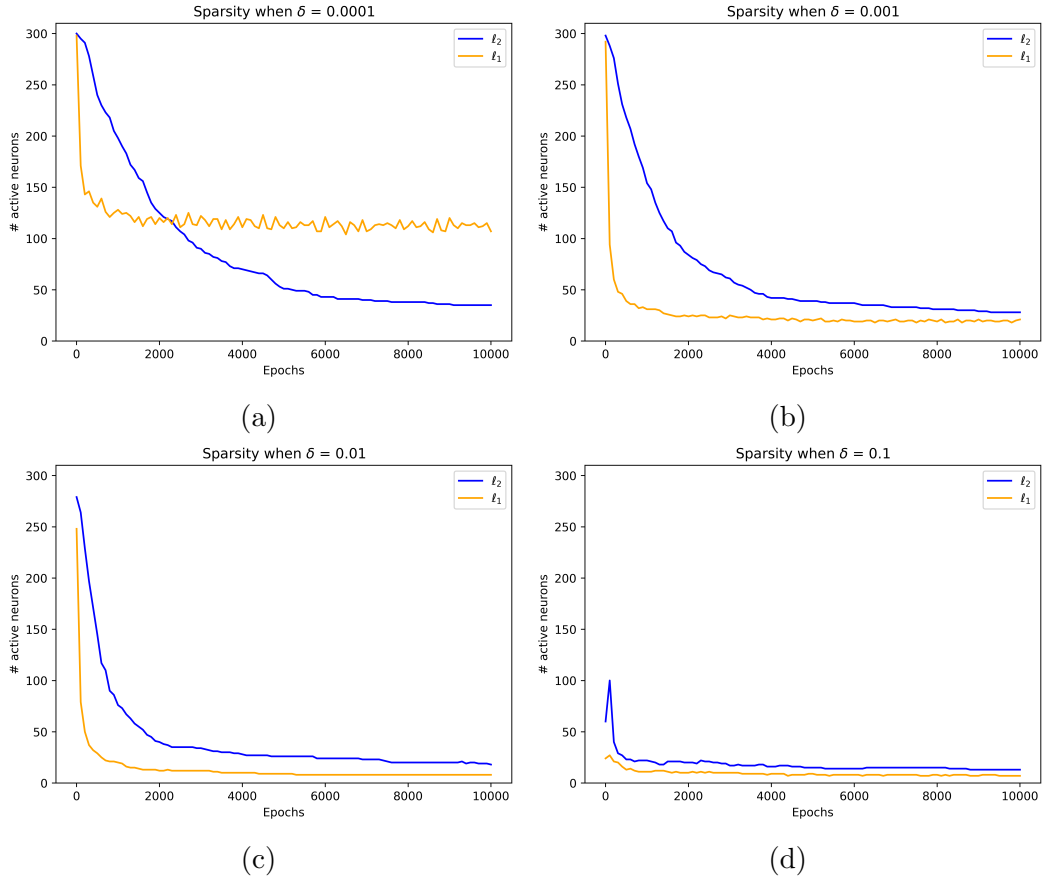


Figure 6.5: Sparsity of ARNN(1) during training, for different thresholds.

6.6 Conclusion

In this chapter, we used Monte-Carlo simulations to study how ARNNs behave in practice with different hyperparameters and architectures. We mainly observed that using penalization enhance the forecasting accuracy of ARNNs for most processes – indeed, the "Saddle" process is better predicted by non-regularized networks. Then, we observed that for nonlinear processes, even if the nonlinearity is very bounded, using both penalization and shortcuts does not improve the predictions and can even degrade if we use the logistic activation function. Finally, we observe a less surprising result for linear autoregressive processes: using a penalized ARNN with shortcuts will yield comparable forecasts with linear AR fit, on average. In short, if nonlinearity of a time series process is highly suspected or confirmed by some non-linearity test, ARNNs will result in better forecasts than ARs, and using penalization ReLU activation functions (and possibly shortcuts in case of slight, very bounded non-linearity) will improve them.

Conclusion & Discussion

Summary of the contributions

In this thesis, we compared a linear forecasting method (i.e. AR(p) fitting) with a nonlinear one (i.e. ARNN(p) with various architectures and hyper-parameters). We reviewed the theoretical framework of both methods. We also reviewed well-known penalization methods, and extended some recent results on the implicit sparsifying properties of ReLU networks with ℓ_2-penalization. In particular, we observed through experiments that these work better than the regular ℓ_1 penalty if one wants to obtain an approximately sparse solution with coefficients closer to real zeros. We finally used Monte-Carlo simulations to study empirically the performances of linear AR(p) fit with ARNN(p) fit. We observed that using penalization enhances the performances of the forecasting model, and that using shortcut connections (i.e. a linear part) does not improve the performances, unless the generating process is itself linear. Finally, we observed empirically that the sparse properties of ReLU networks with ℓ_2-penalization also apply to time-series data fitted by ARNNs.

Future works

In the following section, we present some problems that might be interesting to pursue in future works: these are mainly centred around (1) mathematical proofs for some of the hypothesis and results presented in this thesis, and (2) extending the results to other neural network models. Other possible extensions may be of interest, and are certainly not limited to the tracks we propose. For example, applications to specific fields is of particular interest but we will not give any particular recommendation.

First, based on the results of Leisch et al. (1999), we cannot affirm that ARNNs with ReLU activations are stationary: we can only assume that, in the framework of fitting a predictive model to a stationary process, this activation function is able to approximate a stationary function and the network can converge to a stationary solution. Thus, future work should investigate whether it is possible to prove to which extent and under which conditions ReLU ARNNs

can generate a stationary process.

Another possible track would be to extend these models to the class of deep neural networks (DNNs) which have multiple hidden layers and therefore allow to approximate composed functions. DNNs would allow to approximate processes in which there are discontinuities in the nonlinearity or complex compositions, but can also improve the approximation of smooth functions (Savarese et al., 2019), which is almost impossible to achieve with sparse shallow ReLU networks (because such functions are piecewise linear). Furthermore, recent results from Parhi and Nowak (2022) which are a natural extension of sparse properties in penalized shallow networks to DNNs should be studied in the framework of autoregression in the same fashion as this work.

Some other models, often used in time-series problems, should be studied for NLARMA processes. For example, recurrent neural networks (RNNs), as well as extensions called "long short-term memory" networks (LSTMs) have been around for a while, with many applications: language modelling (Sutskever et al., 2011), speech recognition (Graves et al., 2013), financial time series (Denuit et al., 2019). Other methods which extend convolutional neural networks (CNNs) to time-series data, e.g. WaveNet (Oord et al., 2016), can also be of interest: they do not use recurrent connections and thus are faster than RNNs for long sequences. WaveNet can be used, for example, to generate raw audio (such as speech or music). Extensions to those models based on the sparsity results of this work should be investigated to inspect if they hold for those specific neural models, and to which extent.

Bibliography

Bollerslev, T. (1986). Generalized Autoregressive Conditional Heteroskedasticity. *Journal of Econometrics*, 31(3):307–327.

Box, G. E. P. and Jenkins, G. M. (1976). *Time Series Analysis: Forecasting and Control*. Holden-Day series in time series analysis and digital processing. Holden-Day, San Francisco, rev. ed edition.

Brockwell, P. J. and Davis, R. A. (1991). *Time Series: Theory and Methods*. Springer Series in Statistics. Springer New York, New York, 2nd edition.

Brockwell, P. J. and Davis, R. A. (2016). *Introduction to Time Series and Forecasting*. Springer Texts in Statistics. Springer International Publishing, 3rd edition.

Chollet, F. et al. (2015). Keras. `https://keras.io`.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Math. Control. Signals Syst.*, 2(4):303–314.

Denuit, M., Hainaut, D., and Trufin, J. (2019). *Effective Statistical Learning Methods for Actuaries III: Neural Networks and Extensions*. Springer.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Engle, R. F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica*, 50(4):987–1007.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech Recognition with Deep Recurrent Neural Networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 38:6645—-6649.

Hafner, C. (1998). *Nonlinear Time Series Analysis with Applications to Foreign*

Exchange Rate Volatility. Contributions to economics. Springer.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data mining, Inference and Prediction*. Springer, 2nd edition.

Haykin, S. (2009). *Neural Networks and Learning Machines*. Pearson Education, Upper Saddle River, NJ, third edition.

He, J., Li, L., Xu, J., and Zheng, C. (2020). Relu Deep Neural Networks and Linear Finite Elements. *Journal of Computational Mathematics*, 38(3):502–527.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hoerl, A. E. and Kennard, R. W. (1970). Ridge regression: biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

Hwang, J. T. G. and Ding, A. A. (1997). Prediction intervals for artificial neural networks. *Journal of the American Statistical Association*, 92(438):748–757.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In: 3rd International Conference for Learning Representations, San Diego, 2015.

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). Efficient backprop. In Montavon, G., Orr, G. B., and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade: Second Edition*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lederer, J. (2021). Activation Functions in Artificial Neural Networks: a Systematic Overview. arXiv.

Leisch, F., Trapletti, A., and Hornik, K. (1999). Stationarity and Stability of Autoregressive Neural Network Processes. In Kearns, M. J., Solla, S. A., and Cohn, D. A., editors, *NIPS*, pages 267–273. The MIT Press.

Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feed-

forward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867.

Lu, Z. (1998). On the Geometric Ergodicity of a Non-Linear Autoregressive Model with an Autoregressive Conditional Heteroscedastic Term. *Statistica Sinica*, 8(4):1205–1217.

Masry, E. and Tjøstheim, D. (1995). Nonparametric Estimation and Identification of Nonlinear ARCH Time Series: Strong Convergence and Asymptotic Normality. *Econometric Theory*, 11(2):258–289.

McNeil, A. J., Frey, R., and Embrechts, P. (2005). *Quantitative Risk Management: Concepts, Techniques and Tools*. Princeton UP Series in Finance.

Minsky, M. and Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

Nair, V. and Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In Fürnkranz, J. and Joachims, T., editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814.

Neyshabur, B., Tomioka, R., and Srebro, N. (2014). In Search of the Real Inductive Bias: On the Role of Implicit Regularization in Deep Learning. arXiv.

Nowak, R. D. (2021). What Kinds of Functions Do Neural Networks Learn? [Slides]. Medallion Lecture, JSM 2021, `https://drive.google.com/file/d/1OHjLuJVGFZojVl64VMvEyT24VYYZ9HDL/view`.

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). WaveNet: A Generative Model for Raw Audio. arXiv.

Parhi, R. and Nowak, R. D. (2021). Banach Space Representer Theorems for Neural Networks and Ridge Splines. *Journal of Machine Learning Research*, 22(43):1–40.

Parhi, R. and Nowak, R. D. (2022). What Kinds of Functions Do Deep Neural Networks Learn? Insights from Variational Spline Theory. *SIAM Journal on Mathematics of Data Science*, 4(2):464–489.

Pinkus, A. (1999). Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195.

Reddi, S. J., Kale, S., and Kumar, S. (2019). On the Convergence of Adam and Beyond. arXiv. Published as a conference paper at ICLR 2018.

Rosenblatt, F. (1958). The Perceptron: a Probabilistic Model for Information

Storage and Organization in the Brain. *Psychological Reviews*, 65(6):386–408.

Savarese, P., Evron, I., Soudry, D., and Srebro, N. (2019). How do infinite width bounded norm networks look in function space? In Beygelzimer, A. and Hsu, D., editors, *Proceedings of the Thirty-Second Conference on Learning Theory*, volume 99 of *Proceedings of Machine Learning Research*, pages 2667–2690. PMLR.

Sussmann, H. J. (1992). Uniqueness of the weights for minimal feedforward nets with a given input-output map. *Neural Networks*, 5(4):589–593.

Sutskever, I., Martens, J., and Hinton, G. (2011). Generating Text with Recurrent Neural Networks. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 1017–1024, Madison, WI, USA. Omnipress.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

Tibshirani, R. J. (2021). Equivalences Between Sparse Models and Neural Networks. Available at `https://www.stat.cmu.edu/~ryantibs/papers/sparsitynn.pdf`.

Trapletti, A., Leisch, F., and Hornik, K. (2000). Stationary and Integrated Autoregressive Neural Network Processes. *Neural Comput.*, 12(10):2427–2450.

Verleysen, M. and Lee, J. (2021). LELEC2870: Machine Learning (Regression, Deep Neural Networks and Dimensionality Reduction) [Slides]. Université Catholique de Louvain.

Wackerly, D. D., III, W. M., and Scheaffer, R. L. (2007). *Mathematical Statistics with Applications*. Duxbury Advanced Series, 7th edition.

Yuan, M. and Lin, Y. (2006). Model Selection and Estimation in Regression With Grouped Variables. *Journal of the Royal Statistical Society Series B*, 68:49–67.

Zivot, E. and Wang, J. (2006). *Modeling Financial Time Series with S-Plus*. Springer, New York, NY, 2 edition.

Appendix A

Code for Sparsity Analysis

In this chapter, we provide the Python code to reproduce the results of Section 4.3.3 and 6.5 : one only needs to generate the adequate DGP, be it for classical regression or for time series autoregression. Here, we use the time series experiment.

One needs to import the following libraries. The first block is useful for using convenient arrays and data-frame objects, mathematics functions and visual tools. The second one is necessary to use Keras and therefore build neural network models. Additionally, we import some utility methods to create a custom callback function for sparsity tracking.

Listing A.1: Necessary libraries and imports

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Dense, Input, Add
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1, l2
from tensorflow.keras.callbacks import ModelCheckpoint, Callback
from keras.distribute import distributed_file_utils
from keras.utils import io_utils
from keras.utils import tf_utils
from tensorflow.python.platform import tf_logging as logging
```

Then, we simulate a NLAR process of order 1, defined by Equation (6.1). The object `X` is the vector containing the sequence $(x_t)_{t=1,\dots,T-1}$ and `y` is the vector containing the sequence $(x_t)_{t=2,\dots,T}$.

Listing A.2: Simulation of the nonlinear process, and formatting as convenient Python objects

```
T = 300
burnin = 2000
varepsilon = np.random.normal(0, 0.25, T+burnin)
nlar1 = np.zeros(T+burnin)

def xSim(x):
    return 0.5*(-x + 2*math.exp(-16*(x**2)))

for t in range(1, T+burnin):
    nlar1[t] = xSim(nlar1[t-1]) + varepsilon[t]
nlar1 = nlar1[-T:]

def seriesToDataframe(data, p, n_out=1):
    df = pd.DataFrame(data)
    cols = list()
    # input sequence (t-p, ... t-1)
    for i in range(p, 0, -1):
        cols.append(df.shift(i))
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
    # put it all together
    agg = pd.concat(cols, axis=1)
    # drop rows with NaN values
    agg.dropna(inplace=True)
    return np.asarray(agg)

X = np.asarray(seriesToDataframe(nlar1, p=1, n_out=1))
y = X[:,len(X[0,:])-1]
X = np.delete(X, obj = 1, axis=1)
```

Next, we need to define a custom callback that we name `TrackSparsity`. Its goal is to save the model object to retrieve the weights every e epochs. There already exists a callback in Keras, named `ModelCheckpoint`, to save the model at each epoch, but such a procedure is too time-consuming: we simply rewrite some parts of it in the custom callback. Saving the model every $e = 100$ epochs is more interesting and viable, and one needs to specify the number e with the option `freq`.

Listing A.3: Callback used to track sparsity

```
class TrackSparsity(Callback):

    def __init__(
        self,
        filepath,
        freq
    ):
        super().__init__()
        self._supports_tf_logs = True
        self.freq = freq
        self.filepath = io_utils.path_to_string(filepath)
```

```
def on_epoch_end(self, epoch, logs=None):
    if epoch % self.freq == 0:
        self._save_model(epoch=epoch, batch=None, logs=logs)

def _save_model(self, epoch, batch, logs):
    """Saves the model.
    Args:
        epoch: the epoch this iteration is in.
        batch: the batch this iteration is in. `None` if the `save_freq`
          is set to `epoch`.
        logs: the `logs` dict passed in to `on_batch_end` or
            `on_epoch_end`.
    """
    logs = logs or {}

    filepath = self._get_file_path(epoch, batch, logs)
    self.model.save(
        filepath, overwrite=True
    )
    self._maybe_remove_file()


def _get_file_path(self, epoch, batch, logs):
    """Returns the file path for checkpoint."""

    try:
        # `filepath` may contain placeholders such as
        # `{epoch:02d}`,`{batch:02d}` and `{mape:.2f}`. A mismatch
            between
        # logged metrics and the path's placeholders can cause
            formatting to
        # fail.
        if batch is None or "batch" in logs:
            file_path = self.filepath.format(epoch=epoch + 1, **logs)
        else:
            file_path = self.filepath.format(
                epoch=epoch + 1, batch=batch + 1, **logs
            )
    except KeyError as e:
        raise KeyError(
            f'Failed to format this callback filepath:
                "{self.filepath}". '
            f"Reason: {e}"
        )
    self._write_filepath = distributed_file_utils.write_filepath(
        file_path, self.model.distribute_strategy
    )
    return self._write_filepath

def _maybe_remove_file(self):
    # Remove the checkpoint directory in multi-worker training where
        this
```

```
        # worker should not checkpoint. It is a dummy directory previously
            saved
        # for sync distributed training.
        distributed_file_utils.remove_temp_dir_with_filepath(
            self._write_filepath, self.model.distribute_strategy
        )

# define callbacks with your filepath; within the tmp folder if possible
mysave_w1 =
    TrackSparsity(filepath='C:/tmp/Tracking/model1.{epoch:01d}.h5',
    freq=100)
mysave_w2 =
    TrackSparsity(filepath='C:/tmp/Tracking/model2.{epoch:01d}.h5',
    freq=100)
```

Once times-series have been simulated and the callbacks has been defined, the script is ready for models building. Here, we define common hyper-parameters for both ℓ_2 and ℓ_1 models, and then define those in a "sequential" manner, layer by layer. Finally, both networks are optimized by the AMSGrad algorithm and ready for predictions.

Listing A.4: Model definition and optimization

```
# Hyper-parameters
order_K = T
lambda_L2 = 1e-5
lambda_L1 = 2*lambda_L2
epochs = 10000 + 1
btch_size = 50
alpha_lr = 0.05

### L2-regularization
inputs1 = Input(shape=(1,))
layer1 = Dense(order_K, use_bias=True, activation='relu',
             kernel_regularizer=l2(l2=lambda_L2))(inputs1)
predictions1 = Dense(1, use_bias=True, activation='linear',
                  kernel_regularizer=l2(l2=lambda_L2))(layer1)
model1 = Model(inputs=inputs1, outputs=predictions1)
# optimization
model1.compile(loss='mse',optimizer=Adam(learning_rate=alpha_lr,
    amsgrad=True))
# Training
history1 = model1.fit(X, y, epochs=epochs, batch_size=int(btch_size),
                   verbose=0, callbacks = [mysave_w1])

### L1-regularization
inputs2 = Input(shape=(1,))
layer2 = Dense(order_K, use_bias=True, activation='relu',
               kernel_regularizer=l1(l1=lambda_L1))(inputs2)
predictions2 = Dense(1, use_bias=True, activation='linear',
                    kernel_regularizer=l1(l1=lambda_L1))(layer2)
model2 = Model(inputs=inputs2, outputs=predictions2)
# optimization
```

```
model2.compile(loss='mse',optimizer=Adam(learning_rate=alpha_lr,
    amsgrad=True))
# Training
history2 = model2.fit(X, y, epochs=epochs, batch_size=int(btch_size),
                    verbose=0, callbacks = [mysave_w2])
```

The following blocks are used to visualize the loss during training for both methods, as well as visualization of the networks' fit on the last 30 observations. Note that these observations we used for optimization, but one can easily make adjustments on the code to train on the first (say) 80% and use this kind of plot to visualize forecasting of unseen observations during training.

Listing A.5: Visualization of loss and fit

```
# Loss, Figure 6.3 (b)
plt.figure(figsize=(7,5), dpi=600)
plt.plot(history1.history['loss'], color = "blue")
plt.plot(history2.history['loss'], color = "orange")
plt.legend([r'$\ell_2$', r'$\ell_1$'], loc='upper right')
plt.ylabel('MSE')
plt.xlabel('Epochs')

# Prediction visualization, Figure 6.3 (a)
pred1 = model1.predict(X)
pred2 = model2.predict(X)
# prediction visualization
time_index = np.linspace(251,T,50)
forecast_index = np.linspace(271,T,30)
plt.figure(figsize=(7,5), dpi=600)
plt.plot(forecast_index, pred1[269:T], alpha=0.7, color = "blue")
plt.plot(forecast_index, pred2[269:T], alpha=0.7, color = "orange")
plt.plot(time_index, nlar1[250:T], alpha=0.4, linestyle=":", color =
    "black")
plt.legend([r'$\ell_2$', r'$\ell_1$'], loc='lower left')
plt.xlabel("Time")
plt.ylabel(r"$\{X_t\}$")
plt.ylim((-1.5,1.5))
```

The next block extract the weights of the final model, and use them to plot the sparsity ratio with respect to different δ as shown in Figure 6.4, and then iterates through the saved models to track the sparsity during training. This last block of iterations can be used to reproduce Figure 6.5, and further study the sparse behaviour of regularized networks.

Listing A.6: Visualization of sparsity

```
# extract weights of final model
wgts_out1 = model1.get_weights()[1]
wgts_in1 = model1.get_weights()[0][0]
wgts1 = np.concatenate((wgts_out1,wgts_in1))
wgts_out2 = model2.get_weights()[1]
wgts_in2 = model2.get_weights()[0][0]
```

```
wgts2 = np.concatenate((wgts_out2,wgts_in2))
nwgts = np.linspace(1,len(wgts1),num=len(wgts1))

# Iterate to remake Figure 6.4
active_threshold = []
tshld_vec = np.linspace(0.00001,0.1,10000)
for tsh in tshld_vec:
    tshld = tsh
    active1 = 0
    active2 = 0
    for i in range(len(wgts_in1)):
        active1 += int(abs(wgts_in1[i]) > tshld and abs(wgts_out1[i]) >
            tshld)
        active2 += int(abs(wgts_in2[i]) > tshld and abs(wgts_out2[i]) >
            tshld)
    active_threshold.append(active2/active1)

plt.figure(figsize=(7, 5), dpi=600)
plt.plot(tshld_vec, active_threshold, color = "blue")
plt.ylabel(r'(# $\delta$-active neurons $\ell_1$)/(# $\delta$-active
    neurons $\ell_2$)')
plt.xlabel(r'$\delta$')
plt.xscale('log',base=10)

# Iterate to remake Figure 6.5
tshld_vec = [1e-4,1e-3,1e-2,1e-1]
for tsh in tshld_vec:
    tshld = tsh
    active1 = 0
    active2 = 0
    for i in range(len(wgts_in1)):
        active1 += int(abs(wgts_in1[i]) > tshld and abs(wgts_out1[i]) >
            tshld)
        active2 += int(abs(wgts_in2[i]) > tshld and abs(wgts_out2[i]) >
            tshld)
    print("\nActive neurons:")
    print("L2:", active1, "/", len(wgts_out1))
    print("L1:", active2, "/", len(wgts_out2))

    sparsity1 = []
    sparsity2 = []
    for i in range(1,epochs+100,100):
        track_tmp1 =
            load_model('C:/tmp/Tracking/model1.{epoch}.h5'.format(epoch=i))
        wgts_out1 = track_tmp1.get_weights()[1].round(decimals=6)
        wgts_in1 = track_tmp1.get_weights()[0][0].round(decimals=6)
        sparsity1.append(order_K - np.count_nonzero((abs(wgts_out1) <
            tshld) + (abs(wgts_in1) < tshld)))

        track_tmp2 =
            load_model('C:/tmp/Tracking/model2.{epoch}.h5'.format(epoch=i))
        wgts_out2 = track_tmp2.get_weights()[1].round(decimals=6)
        wgts_in2 = track_tmp2.get_weights()[0][0].round(decimals=6)
```

```
        sparsity2.append(order_K - np.count_nonzero((abs(wgts_out2) <
            tshld) + (abs(wgts_in2) < tshld)))

    epochs_vector = np.linspace(1,epochs,101)
    plt.figure(figsize=(7, 5), dpi=600)
    plt.plot(epochs_vector, sparsity1, color = "blue")
    plt.plot(epochs_vector, sparsity2, color = "orange")
    plt.legend([r'$\ell_2$', r'$\ell_1$'], loc='upper right')
    plt.title(r'Sparsity when $\delta$ = {}'.format(tshld))
    plt.ylabel('# active neurons')
    plt.xlabel('Epochs')
    plt.ylim((0,310))
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
Faculté des sciences
Place des sciences, 2 bte L6.06.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/sc**