People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University of Saida - Dr Moulay Tahar.

Faculty of MIT .

Department of Mathematics.

No. Assigned by the library

This Thesis was Submitted in Partial Fulfillment of the Requirements the degree of

Academic Master

Discipline : MATHEMATICS

Specialty: Stochastic Analysis Statistical of Processes and Applications

by

Mimouna **S**afia **S**ekran[1]

Under the supervision of

Dr. Fatiha Mokhtari

Theme:

Long Short-Term Memory in time series forecasting

Defended on 14 /06/2026 in front of the committee

Dr. M. Kadi	University of Saida Dr. Moulay Tahar	President
Dr. F. Mokhtari	University of Saida Dr. Moulay Tahar	Supervisor
Pr. S. Rahmani	University of Saida Dr. Moulay Tahar	Examiner

June, 2025/2026

[1]e-mail: sekransafia@gmail.com

Contents

List of Figures

Acknowledgments

First, I express my sincere gratitude to Allah, the Almighty, for granting me the health, strength, and perseverance necessary to complete this work. May His blessings be on our Prophet Mohammed.

I express my deepest appreciation and sincere thanks to all those who have contributed, directly or indirectly, to the completion of this dissertation through their support and encouragement.

In particular, I am deeply grateful to my supervisor, **Dr. Fatiha Mokhtari**, for her continuous guidance, patience, availability, trust, and insightful advice throughout the development of this work. Your support and attention were invaluable in answering my numerous questions and in the successful completion of this study.

I also extend my sincere gratitude to the members of the examination committee for the honor they have given me by accepting to evaluate this work.

I am equally grateful to all the professors of the Department of Mathematics for their dedication and for the knowledge they have imparted throughout my academic journey.

I would also like to express my heartfelt appreciation to my family for their unwavering support, encouragement, and understanding throughout my studies.

Furthermore, I wish to extend my sincere thanks to **Ms.Amina Souar**, supervisor at the Algerian Water Company, as well as **Dr. Djdid Saadli**, for providing me with the opportunity to undertake an internship at the Algerian Water Company. I am also grateful to **Mr.Oussama Henni** and **Mr. Zakaria Henni** for their support and assistance in facilitating this internship opportunity that enriched this work.

Finally, I acknowledge with gratitude everyone who has contributed, directly or indirectly, to the completion of this work.

Dedication

To **Allah**, the Most Gracious, the Most Merciful, whose guidance, mercy, and blessings have sustained me throughout this journey. In moments of uncertainty, He granted me strength; in times of difficulty, He blessed me with patience and perseverance. All praise and gratitude belong to Him alone, for without His will and guidance, this work would not have been possible.

I dedicate this dissertation with profound gratitude and sincere appreciation to those whose support, sacrifices, and encouragement made its completion possible.

To my beloved mother,

for her endless love, unwavering faith, and countless sacrifices. Your prayers, patience, and encouragement have been my greatest source of strength. Your constant support has sustained me through every challenge, and your devotion has been the foundation upon which this achievement was built. This work stands as a reflection of all that you have given me.

To my dear father,

for his wisdom, strength, and steadfast belief in my abilities. Your guidance and sacrifices have shaped my path and inspired me to persevere with determination and dignity. Your confidence in me has been a constant source of motivation, and I hope that this achievement is a source of pride for you.

To my grandfather,

with deepest respect and gratitude. Your wisdom, values, and quiet strength have left a lasting mark on my life. Your guidance and the principles you instilled in me have been a source of inspiration throughout this journey. I dedicate this achievement to you with love and appreciation for the support and encouragement you have always provided.

To my brothers and sisters,

for their love, encouragement, and unwavering support. Your presence has been a source of comfort and reassurance throughout this journey. I am deeply grateful for the strength and solidarity you have provided.

To my friends and companions,

for their encouragement, understanding, and support during this academic journey. Your presence, kindness, and shared experiences have provided both comfort and motivation, making the challenges along the way easier to overcome.

To all those who have supported me,

whether through guidance, encouragement, or sincere prayers, I extend my heartfelt gratitude. Every act of kindness and every word of support has contributed to the completion of this work and will always be remembered with appreciation.

This dissertation is not solely the result of individual effort, but also the product of the support, sacrifices, and encouragement of many people whose contributions have been invaluable.

With deepest respect and gratitude,
I dedicate this work to all of you.

“‘

Introduction

Time series forecasting plays a crucial role in many fields, including finance, economics, healthcare, and environmental studies, where the ability to predict future values based on historical data is essential for decision-making. A time series is defined as a set of quantitative observations arranged in chronological order. From a statistical perspective, time series represent realizations of stochastic (aleatory) processes that evolve over time. The fundamental goal of time series modeling is to build a predictive model that produces values close to the observed series, capturing its underlying temporal structure(3).

Time series analysis has attracted significant attention over the past decades, particularly in econometrics, where early developments date back to Jan Tinbergen (1939), who introduced one of the first econometric time series models. Historically, it was assumed that time could be treated as a continuous or discrete variable without strong dependency between observations. Classical linear regression approaches also assumed that residuals are independently and identically distributed. However, in time series data, observations are inherently dependent on previous values, making this assumption invalid in many real-world applications. Depending on their statistical properties, time series can be classified as stationary or non-stationary. Stationary series exhibit constant statistical properties over time, while non-stationary series show evolving mean and variance. Both types can be incorporated into forecasting frameworks, although non-stationary data often require preprocessing such as differencing or transformation(3).

Traditional statistical models, such as autoregressive and moving average methods, have been widely used for forecasting tasks. However, these approaches often struggle to capture complex nonlinear patterns and long-term dependencies present in real-world data. In recent years, advances in deep learning have introduced powerful alternatives for sequential data modeling. Among these methods, Long Short-Term Memory (LSTM) networks, a specialized type of recurrent neural network (RNN), have gained significant attention due to their ability to learn both short-term and long-term temporal dependencies. By incorporating memory cells and gating mechanisms, LSTM networks effectively overcome the limitations of traditional RNNs, such as the problem of the vanishing gradient(13)(18)(20).

This dissertation focuses on the application of LSTM networks for time series forecasting. The main objective is to investigate their effectiveness in modeling temporal dynamics and to evaluate their performance compared to classical forecasting techniques. Through experimental analysis on real-world datasets, this work aims to highlight both the advantages and limitations of LSTM-based approaches in time series prediction tasks.

The initial chapter lays the foundation for understanding time series analysis and forecasting. It introduces the concept of time series, presents illustrative examples, and highlights their applications in various domains. The chapter also discusses the funda-

mental components of a time series, including trend, seasonality, and residual variations. In addition, it explains the concepts of stationarity and autocorrelation, which are essential for time series modelling. Classical stochastic models such as AR, MA, ARMA, ARIMA, and SARIMA processes are presented, along with the distinction between stationary and non-stationary processes. Finally, the chapter introduces the main forecasting approaches, including exponential smoothing and the Box–Jenkins methodology.

The second chapter introduces the fundamental concepts of artificial intelligence, machine learning, and deep learning, which constitute the theoretical foundation of modern predictive models. It begins with an overview of artificial intelligence, including its definition, historical development, different types, and major applications. The chapter then presents machine learning, its main categories, learning process, and limitations. Furthermore, the chapter explores deep learning as an advanced branch of machine learning, highlighting its principles, advantages, and differences from traditional machine learning methods. It also explains the transition from biological neurons to artificial neurons and introduces the structure and functioning of artificial neural networks. Special attention is given to recurrent neural networks (RNNs), their architecture, operating principles, applications, and limitations in time series forecasting. Finally, the chapter concludes with a comparison between artificial neural networks and recurrent neural networks.

The third chapter focuses on the use of Long Short-Term Memory (LSTM) networks for time series forecasting. It begins by introducing the definition, advantages, and applications of LSTM models. The chapter then presents the architecture of LSTM networks, detailing the different components of the LSTM cell, including the input gate, output gate, and cell state, along with their mathematical formulations and functional roles.

In addition, the chapter explains the working mechanism of LSTM networks and analyzes the behavior of each gate during the learning process. A practical methodology for implementing LSTM models is also provided through a step-by-step procedure, including data preparation, model construction, training, evaluation, and prediction generation. Furthermore, the chapter discusses the application of LSTM models to time series forecasting and compares their performance with traditional statistical models such as ARIMA. Finally, illustrative examples and an application on real-world data are presented to demonstrate the effectiveness of LSTM networks in forecasting tasks.

Chapter 1

Overview of Time Series

The world as we know it is filled with time-dependent phenomena. A time series is a sequence formed by observations over time. Time series analysis is a commonly used tool today. This field has many applications in finance, medicine, econometrics, meteorology, and many other areas, for example, in finance, we are interested in modeling the exchange rate of a currency. In meteorology, scientists model the maximum temperature in Alger over the past year to predict the maximum temperature for the upcoming month.

The idea is to take a sample of data and build the best model that fits these data. This model allows us to drow certain peaks, or model the trend(direction)of the series. Another important aspect of the series is the seasonal component, which refers to the presence of cycles. Another interesting concept would be the phenomenon of causality, that is, the influence of one series on another.

One of the main objectives of studying a time series is forecasting, often for economic reasons(predicting the evolution of sales of a certain product, forecasting electricity consumption to optimally adjust production). Forecasts are made based on what has happened before the moment from which the forecast starts. sometimes, additional information is also used. The particularity of time series lies in the presence of a chronological relationship between times, which organizes all the information. In forecasting sales, for example , for the year T+1, one would rely on the evolution of sales during the years T,T-1,....., but also take into account potentially exogenous indices(economic conditions, certain unexpected recent events, relationships between temperature, wind speed, and humidity level,etc.).

However, it is not always easy to choose the right model, as several may normally be good candidates. Most of these will tend to minimize the variance of the residuals. The number of parameters to estimate must also be considered. Normally, we use the principal of parsimony , which suggests choosing the model that requires the fewest parameters while maintaining a low variance of the residuals. Therefore, there are several criteria to consider in creating a model.

There is always a forecasting error, and good methods provide not just a forecast, but

a forecast interval. It often happens that a minimal improvement in the quality of the forecast has a significant impact on costs. Among the most commonly used forecasting methods are:

- Exponential smoothing is a set of extremely simple procedures to implement. They are covered in this chapter.
- The most popular forecasting methods for the past thirty years are those related to ARMA modeling .They essentially consist of identifying the obvious trends in the phenomenon we observe (growth, periodicity, etc.)and focusing on what remains after they have been removed. We model the obtained residual and rely on this modeling to obtain the forecast. These methods, more sophisticated but more effective than exponential smoothing, are costly in terms of computation time. Their systematic use in all fields, whether economic, industrial, physical, etc,..?is due to the exponentially rapid development of computing tools. The art of the statistician, when faced with difficult data, is to confront several methods and choose the one that fits best.

This chapter provide the necessary definitions for understanding the body. Then present a description of the different components, and their approximations. Exponential smoothing and ARMA modeling.

1.1 Definition of a Time Series

A time series is a sequence of observations of the same entity taken at regular time intervals such as daily, monthly or yearly measurements, time series data can be used to make informed predictions of future values.

1.2 Examples

Some common examples of time series data include:

- The monthly unemployment percentage
- The electrical activity of a heart measured in milliseconds (EKG (Electrocardiogram)).
- The quantity of a specific product in stock at the end of each week.
- The daily flow rate of a river in cubic meters per second.
- The number of births per 1,000 people in a country per year.

- The daily ticket sales for a film after its release.
- The number of reported burglaries in a city per month.
- The total tonnage of cargo handled by a major port per month.
- co2 dataset, time series consist of monthly atmospheric co2 concentrations (in parts per million ppm) measured at Mauna Loa Observatory from 1959 to 1997.

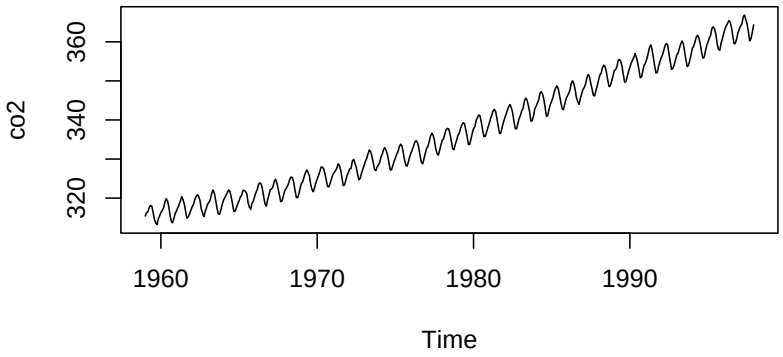


Figure 1.1: The co2 Time Series Data

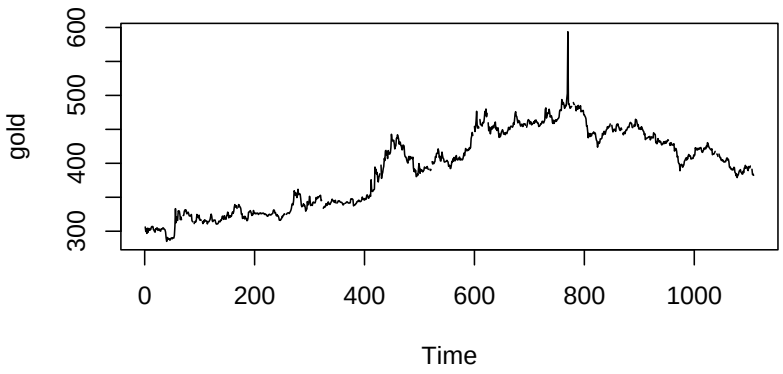


Figure 1.2: Daily gold prices in US dollars. 1 January 1985 – 31 March 1989.

1.3 Applications

Time series analysis has numerous applications across various fields, the following list is just a sample:

- Finance and econometrics : evolution of stock indices, prices, economic data of companies, sales and purchases of goods, agricultural or industrial productions.
- Insurance: analysis of claims.
- Data processing : successful measurements of the position or direction of a moving object(trajectory tracking).
- Signal processing: communication signals, radar signals, sonar signals, speech analysis.

1.4 Fundamental Components of Time Series

1.4.1 The Trend

The trend component c_t represents the long-term evolution, it shows whether the data are generally increasing, decreasing, or remaining stable over a long period. For example, if sales increase every year, the series has an upward trend. If temperatures gradually rise over decades, this reflects a positive long-term trend.

1.4.2 The Seasonal Component

s_t represents repeated patterns in the data that occur at regular time intervals. These fluctuations are typically associated with seasonal effects caused by natural factors, such as weather conditions, or by economic and social activities, such as holidays and promotions. In general, s_t captures movements that recur in a similar way during the same period each year (e.g., monthly or quarterly).

1.4.3 The Residual Component

ϵ_t captures irregular fluctuation that are typically random in nature we suppose that:

- $\mathbb{E}[\epsilon_t] = 0$
- $\text{Var}(\epsilon_t) = \sigma^2$
- $Cov(\epsilon_t, \epsilon_s) = 0 \quad \text{for } t \neq s$

1.4.4 The Accidental Phenomenon

are abnormally high or low isolated values of short duration. These sudden variation on the series are generally explicable(Feb 2022, crises between Russia and Ukran, financial crash, etc,..). Most of the time the accidents are integrated into the white noise.

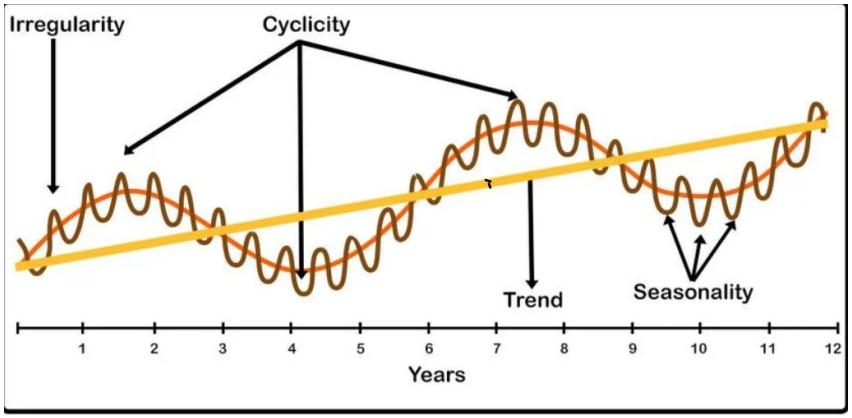


Figure 1.3: Time Series Components

1.5 Stationary

1.5.1 Definitions

Throughout this section, we will consider $(X_t)_{t\in\mathbb{Z}}$ and will assume $X_t \in L^2(\Omega, \mathcal{A}, P)$ for all $t \in \mathbb{Z}$.

Strict or Strong Stationary

The process $(X_t)_{t\in\mathbb{Z}}$ is said to be strictly stationary if:

$\forall n \in \mathbb{N}, \forall (t_1, \ldots, t_n), \forall h \in \mathbb{Z},$ the law of $(X_{t_1}, \ldots, X_{t_n})$ is identical to the law of $(X_{t_1+h}, \ldots, X_{t_n+h})$.

Kolmogorov's Theorem

The process $(X_t)_{t\in\mathbb{Z}}$ is strictly stationary if and only if the law of $(X_t)_{t\in\mathbb{Z}}$ is identical to the law of $(Y_t)_{t\in\mathbb{Z}}$ where $Y_t = X_{t+h}$.

Weak Stationary

The process $(X_t)_{t\in\mathbb{Z}}$ is said to be a second order stationary process (or weakly stationary) if it satisfies:

(i) $\forall t \in \mathbb{Z}, E(X_t) = m$

(ii) $\forall t \in \mathbb{Z}, V(X_t) = \sigma^2 = \text{const}$

(iii) $\forall t \in \mathbb{Z}, \forall h \in \mathbb{Z}, \text{Cov}(X_t, X_{t+h}) = \gamma(h)$ (depends only on h)

where $\gamma(h)$ is the autocovariance of order h of X_t.

Example of Stationary Processes

1. Weak white noise $(\epsilon_t)_{t\in\mathbb{Z}}$ if and only if:

$$E(\epsilon_t) = 0, \quad \forall t \in \mathbb{Z}$$

$$V(\epsilon_t) = \sigma^2, \quad \forall t \in \mathbb{Z}$$

$$\text{Cov}(\epsilon_t, \epsilon_s) = 0, \text{ if } t \neq s$$

   We denote $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$.

2. ϵ_t is strong white noise if and only if the ϵ_t are i.i.d., $E(\epsilon_t) = 0$ and $V(\epsilon_t) = \sigma^2$.

3. First order moving average process, denoted MA(1): Let $\theta \in \mathbb{R}$. Let $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. Let $(X_t)_{t\in\mathbb{Z}}$ defined by: $\forall t \in \mathbb{Z}, X_t = \epsilon_t - \theta\epsilon_{t-1}$. Then $(X_t)_{t\in\mathbb{Z}}$ is a stationary process. We say that X_t

Example (Non-Stationary Processes)

1. Random Walk: Let $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. The process $(X_t)_{t\in\mathbb{Z}}$ is a random walk without drift if and only if:

   (i) $X_t = X_{t-1} + \epsilon_t, \quad \forall t > 0$

   (ii) $\text{Cov}(\epsilon_t, X_{t-k}) = 0, \quad \forall 0 < k \leq t.$

   Even though we have the property $E(X_t) = E(X_{t-1}) = m$, for all $t \in \mathbb{Z}$, the process $(X_t)_{t\in\mathbb{Z}}$ is not stationary:

$$X_t = X_{t-1} + \epsilon_t$$

$$X_{t-1} = X_{t-2} + \epsilon_{t-1}$$

$$\vdots$$

$$X_1 = X_0 + \epsilon_1$$

   Therefore, we have:

$$X_t = X_0 + \sum_{k=1}^{t} \epsilon_k$$

   Thus, the variance evolves as follows:

$$V(X_t) = V(X_0) + 2\sum_{k=1}^{t} \text{Cov}(\epsilon_k, X_0) + V\left(\sum_{k=1}^{t} \epsilon_k\right) = V(X_0) + t\sigma^2.$$

   Consequently, the process is not stationary in variance.

1.5.2 Autocovariance Function and Autocorrelation Function

- The autocovariance of a stationary process $(X_t)_{t\in\mathbb{Z}}$ is defined by:

$$\gamma : \mathbb{Z} \to \mathbb{R}, \quad h \mapsto \gamma(h) = \mathrm{Cov}(X_t, X_{t-h})$$

- The autocorrelation function of a stationary process $(X_t)_{t\in\mathbb{Z}}$ is defined by:

$$\rho(h) = \frac{\gamma(h)}{\gamma(0)} = \mathrm{Corr}(X_t, X_{t+h}) \quad \forall h \in \mathbb{Z}$$

1.6 Stationary Process(ARMA process)

ARMA are stationary processes, and ARIMA are non-stationary integrated processes, i.e., they are made stationary by differencing.

1.6.1 Autoregressive Processes

Definition and Canonical Representation

Definition 2.1.1 (AR Process): $(X_t)_{t\in\mathbb{Z}}$ is an AR(p) process if

(i) (X_t) is stationary

(ii) X_t satisfies the equation

$$X_t = \mu + \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \varepsilon_t$$

with $\phi_p \neq 0$ and ε_t white noise (mean 0, finite variance, and uncorrelated).

We write $\Phi(L)X_t = \mu + \varepsilon_t$ where $\Phi(L) = 1 - \phi_1 L - \cdots - \phi_p L^p$.

Example 2.1.1: AR(1), i.e. $(1 - \psi L)X_t = \mu + \varepsilon_t$ with $|\psi| < 1$ and ε_t white noise.

1.6.2 Moving Average Process

Definition 2.2.1 (MA Process): $(X_t)_{t\in\mathbb{Z}}$ is an MA(q) process if there exist a white-noise sequence $\{\varepsilon_t\}$ with

$$\mathbb{E}[\varepsilon_t] = 0, \quad \mathrm{Var}(\varepsilon_t) = \sigma^2, \quad \mathbb{E}[\varepsilon_t \varepsilon_s] = 0 \text{ for } t \neq s,$$

and coefficients $\theta_1, \ldots, \theta_q$ such that

$$X_t = \mu + \varepsilon_t - \theta_1 \varepsilon_{t-1} - \cdots - \theta_q \varepsilon_{t-q}.$$

1.6.3 ARMA Processes (AutoRegressive Moving Average)

A process X_t is ARMA(p, q) if it can be expressed as a combination of past values (AR) and past shocks (MA):

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}, \tag{1.1}$$

where ε_t is a white-noise sequence.

1.7 Non Stationary process

1.7.1 ARIMA Process

For a time series X_t, an ARIMA(p, d, q) model can be written as:

$$(1 - \phi_1 L - \ldots - \phi_p L^p)(1 - L)^d X_t = \mu + (1 - \theta_1 L - \ldots - \theta_q L^q)\varepsilon_t \tag{1.2}$$

Where:

- L is the backshift operator: $B^k X_t = X_{t-k}$
- ϕ_i are autoregressive parameters for $i = 1, \ldots, p$
- θ_j are moving average parameters for $j = 1, \ldots, q$
- ε_t is white noise with $\varepsilon_t \sim \text{WN}(0, \sigma^2)$
- c is a constant term

1.7.2 SARIMA Process

The **SARIMA** (Seasonal Autoregressive Integrated Moving Average) model, denoted as

$$\text{SARIMA}(p, d, q)(P, D, Q)_s,$$

is an extension of the ARIMA model that incorporates seasonality. It is defined as:

$$\Phi_P(B^s)\,\phi_p(B)\,(1 - B)^d\,(1 - B^s)^D\,y_t = \Theta_Q(B^s)\,\theta_q(B)\,\varepsilon_t,$$

where:

- y_t is the observed time series at time t;
- ε_t is a white noise process with mean zero and variance σ^2;
- B is the backshift operator such that $B^k y_t = y_{t-k}$;

- s is the number of periods per season (e.g., $s = 4$ for quarterly data, $s = 12$ for monthly data);
- p is the order of the non-seasonal autoregressive (AR) part;
- d is the degree of non-seasonal differencing;
- q is the order of the non-seasonal moving average (MA) part;
- P is the order of the seasonal autoregressive part;
- D is the degree of seasonal differencing;
- Q is the order of the seasonal moving average part.

The polynomials are defined as:

$$\begin{aligned}
\phi_p(B) &= 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p, \\
\theta_q(B) &= 1 + \theta_1 B + \theta_2 B^2 + \cdots + \theta_q B^q, \\
\Phi_P(B^s) &= 1 - \Phi_1 B^s - \Phi_2 B^{2s} - \cdots - \Phi_P B^{Ps}, \\
\Theta_Q(B^s) &= 1 + \Theta_1 B^s + \Theta_2 B^{2s} + \cdots + \Theta_Q B^{Qs}.
\end{aligned}$$

Equivalently, the model can be written in expanded form as:

$$\begin{aligned}
&(1 - \phi_1 B - \cdots - \phi_p B^p)(1 - \Phi_1 B^s - \cdots - \Phi_P B^{Ps})(1 - B)^d (1 - B^s)^D y_t \\
&\quad = (1 + \theta_1 B + \cdots + \theta_q B^q)(1 + \Theta_1 B^s + \cdots + \Theta_Q B^{Qs}) \varepsilon_t.
\end{aligned}$$

1.8 Time Series Forecasting

Time series forecasting is the process of predicting future values using historical data collected over time. Unlike regular prediction problems, time series data is ordered chronologically, and past values influence future values. Common forecasting methods include:

- Statistical models like Exponential Smoothing and ARIMA
- Machine learning models such as regression and neural networks

The statistical models will be presented in this section.

1.8.1 Exponential Smoothing

A number of techniques are grouped under the term "Exponential Smoothing". These approaches provide fairly accurate forecasts. they are low-cost, and calculations can be performed very quickly using recursive update formulas.

These techniques were introduced by Holt in 1958, Winters in 1960, and popularized by Brown's book in 1963. Smoothing methods encompass a set of empirical forecasting techniques that assign varying importance to past values of a time series.

Additive Holt-Winters Model

Additive Holt–Winters smoothing (also called triple exponential smoothing) is a forecasting method used for time series data that contains:A trend, Seasonality and Relatively constant seasonal variations (additive seasonality) The additive Holt–Winters model has three components:

- Level (L_t) – baseline value
- Trend (T_t) – growth or decline over time
- Seasonal (s_t) – repeating seasonal pattern

Forecast Equation:

The h-step-ahead forecast is given by:

$$\hat{y}_{t+h} = L_t + hT_t + S_{t+h-mk}$$

where:

- m is the length of the seasonal cycle,
- h is the forecast horizon,
- $k = \left\lfloor \frac{h-1}{m} \right\rfloor$ ensures proper seasonal indexing.

Updating Equations:

At each time step t, the components are updated as follows:

Level

$$L_t = \alpha(y_t - S_{t-m}) + (1-\alpha)(L_{t-1} + T_{t-1})$$

Trend

$$T_t = \beta(L_t - L_{t-1}) + (1-\beta)T_{t-1}$$

Seasonal

$$S_t = \gamma(y_t - L_t) + (1-\gamma)S_{t-m}$$

where:

- α is the level smoothing parameter,

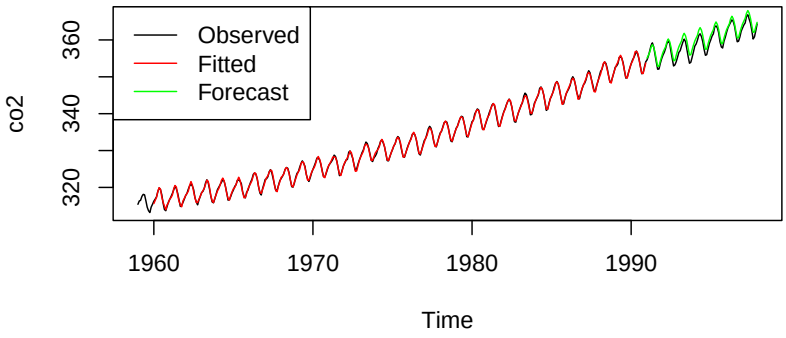


Figure 1.4: co2 with Holt-Winters Fit

- β is the trend smoothing parameter,
- γ is the seasonal smoothing parameter,
- $0 < \alpha, \beta, \gamma < 1$.

Multiplicative Holt-Winters Model

The Multiplicative Holt-Winters Model is a popular forecasting method used for time series data that exhibits both a trend and seasonal pattern, where the magnitude of the seasonal fluctuations grows or shrinks proportionally with the level of the series.

The model consists of three smoothing equations and one forecast equation.

Let:

y_t = actual value at time t

m = number of periods in a season (e.g., 12 months)

α = smoothing factor for the level $(0 < \alpha < 1)$

β = smoothing factor for the trend $(0 < \beta < 1)$

γ = smoothing factor for the seasonality $(0 < \gamma < 1)$

The model is defined by three smoothing equations:

$$L_t = \alpha \frac{y_t}{S_{t-m}} + (1-\alpha)(L_{t-1} + T_{t-1}) \tag{1.3}$$

The actual value is deseasonalized by dividing by the most recent estimate of the seasonal factor for that period, S_{t-m}.

$$T_t = \beta(L_t - L_{t-1}) + (1-\beta)T_{t-1} \tag{1.4}$$

The trend is updated as the difference between the newly estimated level and the previous level.

$$S_t = \gamma \frac{y_t}{L_t} + (1 - \gamma) S_{t-m} \tag{1.5}$$

The seasonal factor is updated as the ratio of the actual value to the current estimated level.

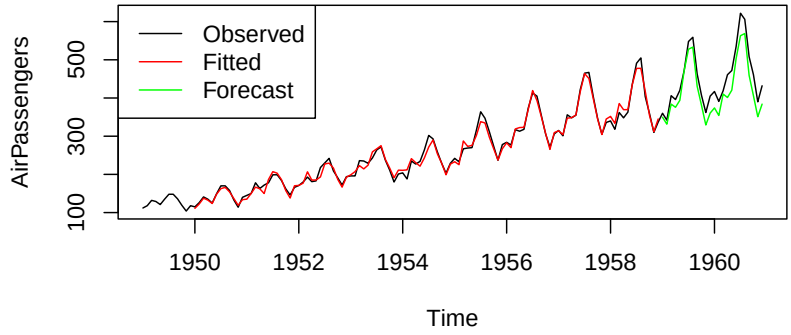


Figure 1.5: AirPassengers with Holt-Winters Fit

Forecast Equation

The forecast for k periods ahead, made from time t, is given by:

$$F_{t+k} = (L_t + k \times T_t) \times S_{t+k-m} \tag{1.6}$$

The forecast is the sum of the level and the extrapolated trend, multiplied by the appropriate seasonal factor from the last seasonal cycle.

1.8.2 Box and Jenkins Method

The Box and Jenkins method is a systematic approach to identifying, estimating and diagnosing ARIMA models. it's developed by George Box and Gwilym Jenkins in 1970, in order to build an optimal time series model for forecasting(3) .

This methodology suggests three steps:

step 1: Model Identification

The initial step involves examining the properties of the time series to determine the tentative orders of the ARIMA model, denoted as ARIMA(p, d, q). The primary tasks in this step are:

1. **Stationarity Assessment:** The first step is to check whether the time series is stationary, meaning its mean, variance, and autocorrelation structure are constant over time. This is typically done by visual inspection of the time series plot and by applying formal statistical tests such as the Augmented Dickey-Fuller (ADF) test.

2. **Differencing:** If the series is found to be non-stationary, it must be transformed to achieve stationary. The most common transformation is *differencing*, where the current value is subtracted from the previous value. The number of times differencing is applied to achieve stationary determines the integrated component, d, of the ARIMA model.

3. **Order Selection:** Once a stationary series is obtained, the autocorrelation function (ACF) and partial autocorrelation function (PACF) are examined. The patterns observed in these correlograms provide clues for tentatively selecting the order of the autoregressive (AR) component, p, and the moving average (MA) component, q. For instance, a sharp cut-off in the PACF and a decaying tail in the ACF suggests an AR model, while the opposite pattern suggests an MA model.

step 2: Parameter Estimation

After tentatively identifying the orders (p, d, q), the next stage focuses on estimating the parameters (coefficients) of the specified model. Using the available time series data, numerical optimization techniques are employed to find the parameter values that provide the best fit. The most commonly used method is **maximum likelihood estimation (MLE)**, which finds the parameter values that maximize the likelihood of observing the given data. Other methods, such as non-linear least squares, may also be used. The output of this stage is a fully specified model with estimated coefficients.

step 3: Diagnostic Checking

The final core stage involves verifying the adequacy of the estimated model by examining its residuals, $\hat{\varepsilon}_t$ (the difference between the actual and fitted values). A key assumption of the ARIMA model is that the residuals should behave as a **white noise** process. That is, they should be independently and identically distributed with a mean of zero, constant variance, and no autocorrelation.

1. **Residual Analysis:** The residuals are examined by plotting their ACF and PACF. If the model is adequate, these plots should show no significant autocorrelations at any lag.

2. **Statistical Tests:** Formal statistical tests, such as the **Ljung-Box Q-test**, are applied to test the joint hypodissertation that a group of autocorrelations of the

residuals is zero. A non-significant test statistic (p-value > 0.05) suggests that the residuals are indeed white noise.

3. **Iteration:** If the diagnostic checks reveal that the residuals are not white noise (e.g., significant autocorrelations are present), the model is deemed inadequate. In this case, the researcher must return to the **Identification** stage (Section 1.8.2) to tentatively select a new, more appropriate model. The process of estimation and diagnostic check is then repeated. This cycle continues until a statistically adequate model is found.

Forecasting

Once a model has passed all diagnostic checks and is considered adequate, it can be used for its ultimate purpose: forecasting. The final model is used to generate predictions for future time periods, along with appropriate confidence intervals that reflect the uncertainty associated with the forecasts.

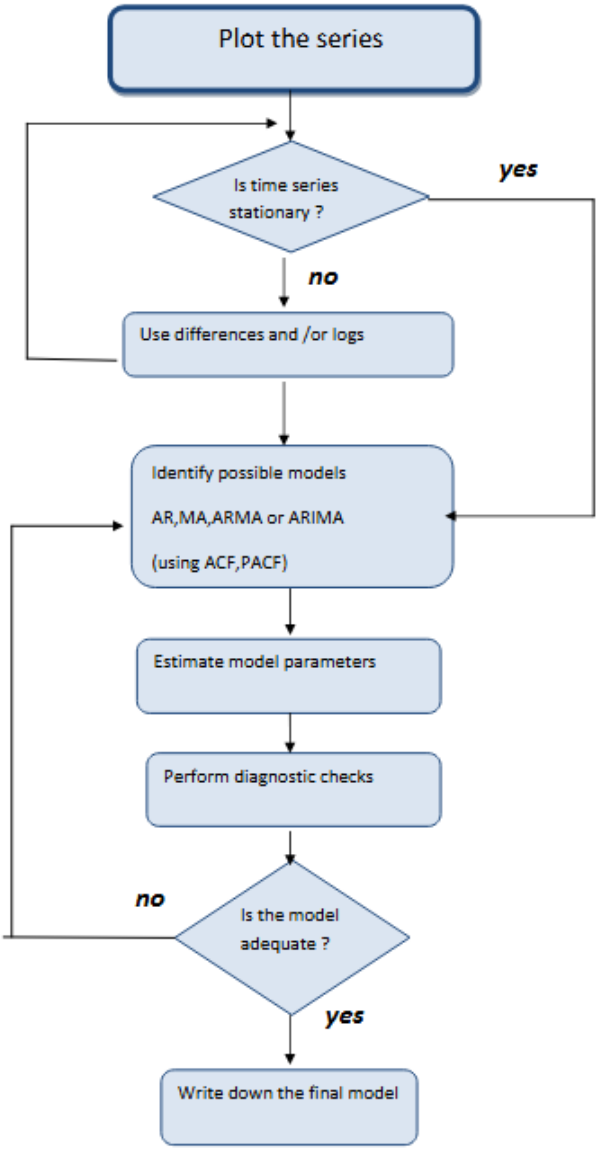


Figure 1.6: Box-jenkins Method

This is an example of the milk dataset, which represents monthly milk production over several years..

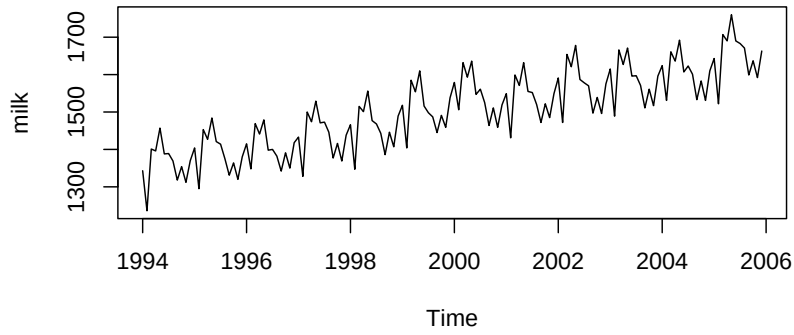


Figure 1.7: Average Monthly Milk Production Per Cow In The US, 01/1994 - 12/2005

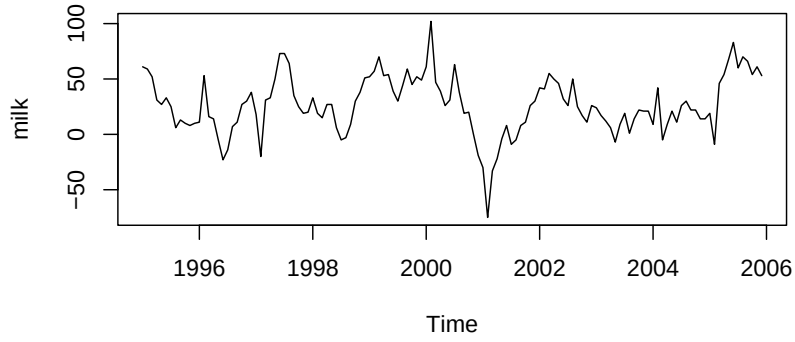


Figure 1.8: Difference of The Series

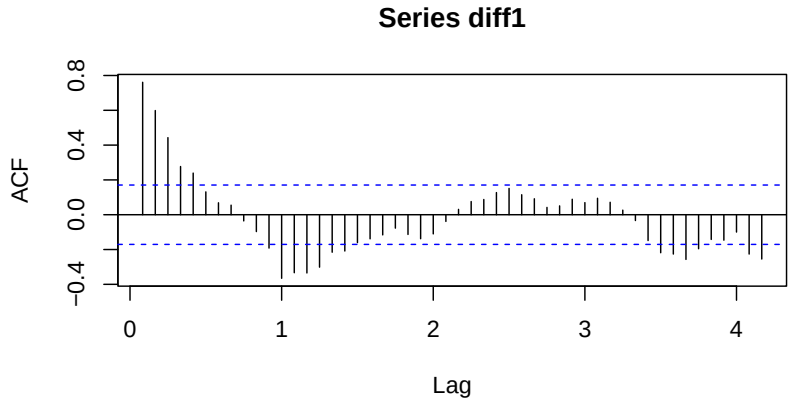


Figure 1.9: ACF of The Differenced Series

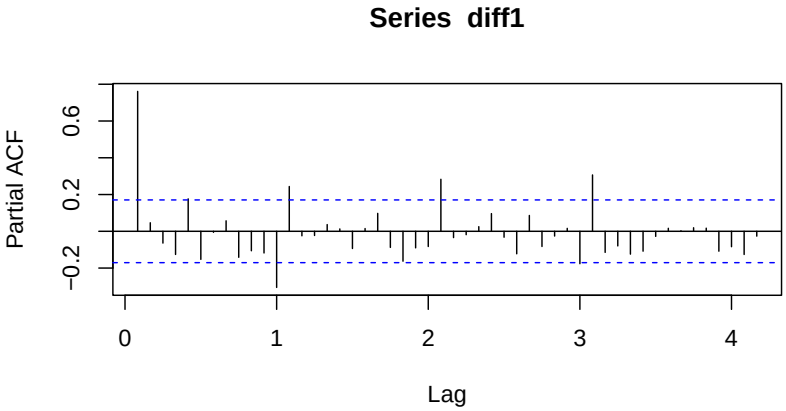


Figure 1.10: PACF of The Differenced Series

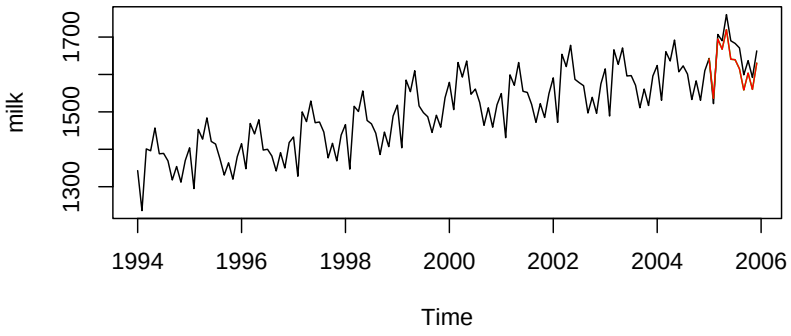


Figure 1.11: Prediction of The Last Year

Chapter 2

Introduction to Deep Learning and Neural Networks

Artificial Intelligence (AI) has emerged as one of the most influential fields in modern science and technology. Its primary objective is to design systems capable of performing tasks that typically require human intelligence, such as learning, reasoning, pattern recognition, and decision-making (25). Over the past decades, (AI) has evolved from rule-based symbolic systems to data-driven approaches that rely on statistical and computational learning methods.

Machine Learning (ML), a core subset of artificial intelligence, enables systems to automatically learn patterns from data without being explicitly programmed (1). By leveraging mathematical models and optimization techniques, machine learning algorithms can generalize from historical data to make predictions or informed decisions. These methods have demonstrated remarkable success across various domains including finance, healthcare, engineering, and economics.

Among machine learning techniques, Deep Learning (DL) represents a significant advancement (13). Deep learning models, also known as deep neural networks, are based on artificial neural networks with multiple hidden layers, allowing them to learn hierarchical feature representations from complex and high-dimensional data . Their ability to model nonlinear relationships has made them particularly effective in image processing, speech recognition, natural language processing, and increasingly, time series forecasting.

Deep learning has shown particular promise, time series forecasting stands out. Time series data present unique challenges due to their sequential and temporal dependencies. Traditional statistical models often rely on linear assumptions that may not adequately capture complex dynamic patterns.

This chapter is organized as follows. Section 2.1 provides an overview of artificial intelligence and its evolution. Section 2.2 introduces machine learning, covering key paradigms and learning strategies. Section 2.3 discusses deep learning, focusing on neural network architectures and their advantages.

2.1 Artificial Intelligence

2.1.1 Definition

Artificial Intelligence refers to the ability of machines to simulate human intelligence processes such as learning, reasoning, and decision-making. It aims to develop systems capable of performing tasks that normally require human intelligence (25).

2.1.2 Types of Artificial Intelligence

AI can be categorized into the following types:

- **Narrow AI**: Designed to perform a specific task (e.g., speech recognition).
- **General AI**: Capable of performing any intellectual task that a human can do.

2.1.3 Applications of AI

AI is widely used in various domains, including:

- Finance (stock market prediction)
- Healthcare (disease diagnosis)
- Transportation (autonomous vehicles)
- Forecasting and decision-making systems

2.2 Machine Learning

2.2.1 Introduction

Machine Learning is something we all use hundreds of times each day without even realizing it. Take a simple Google search, for example. One reason we get relevant results is that a Machine Learning algorithm works behind the scenes, having learned how to identify the most appropriate results from billions of possibilities. This type of learning enables a system to continuously improve by adapting to its environment, past

experiences, and observed outcomes. Achieving this with traditional programming would be nearly impossible, as it would require writing code for an endless number of specific cases. However, with Machine Learning, we give algorithms the ability to learn on their own, which makes solving complex problems far more manageable.

This section serves as our first step into the field of machine learning. It will provide a general understanding of how a machine can learn to produce reasonably reliable results.

2.2.2 History

Machine Learning emerged in the second half of the 20th century from the field of AI.

- The first traces of AI date back to 1950 in an article by Alan Turing titled "Computing Machinery and Intelligence," in which the mathematician explores the problem of defining whether a machine is conscious or not.
- In 1957, another American computer scientist, Frank Rosenblatt, created the "perceptron" – an algorithm that constitutes the first artificial neural network.
- In 1959, American computer scientist Arthur Samuel used the term "Machine Learning" for the first time for his program created in 1952.
- Then came a significant halt for AI and Machine Learning; so much so that we speak of an "AI winter" starting in 1974. Although the technology saw a rebound in the early 1980s, it faced a new crisis from 1987 to 1993 – a period considered the "second AI winter."
- The early 1990s marked the revival of Machine Learning.
- In the early 2000s, artificial intelligence research changed its paradigm and went to machine learning. Until then, to teach something to a machine, you had to manually encode every rule to be followed.
- In 2012, a neural network developed by Google managed to recognize human faces as well as cats in YouTube videos.
- In 2016, an artificial intelligence system based on machine learning called LipNet succeeded in lip-reading with a high success rate (2). All of this leads to the Machine Learning of today.

2.2.3 Definition

Machine Learning is a subset of Artificial Intelligence that enables systems to learn from data and improve their performance without being explicitly programmed (1). It is both a science and an art of programming computers to learn directly from data, rather than following explicit instructions (9). Stated informally, it is the field concerned with enabling computers to acquire knowledge and make predictions or decisions from data.

2.2.4 Important Terms

- **Dataset:** the data am going to use as an initial input for my model.
- **Features:** the unique attributes about your dataset.
- **Data Labeling:** the target variable am trying to predict.
- **Training & Testing Data:** how you split your data set to train and test your model results.
- **Algorithm:** the tool you choose to use specific to the problem you are solving.

2.2.5 Types of Machine Learning

Machine Learning algorithms are generally classified into the following categories:

- **Supervised Learning**: The model is trained using labeled data. In supervised learning, the training dataset provided to the algorithm includes the correct answers, known as labels (9). The algorithm learns to build a predictive model by studying many pairs of inputs (features) and their corresponding outputs (labels). The labels act as a guide, supervising the learning process. The ultimate aim is for the model to generalize accurately to new, unseen data.

  Supervised learning is commonly divided into:

  - **Regression:** Predicting a continuous numerical value, such as house prices.
  - **Classification:** Predicting a discrete category, such as whether an email is spam or not.

  Important algorithms in this category include linear regression, logistic regression, support vector machines (SVMs), decision trees, and random forests (9).

- **Unsupervised Learning**: The model identifies patterns in unlabeled data. Unsupervised learning works with data that has no labels, meaning no teacher provides correct answers (9). The algorithm aims to build descriptive models that uncover

hidden patterns, groupings, or structures within the data on its own. It is often used for exploratory data analysis.

Key techniques in unsupervised learning include:

- **Clustering:** Automatically grouping similar data points together (e.g., customer segmentation).
- **Dimensionality reduction:** Reducing the number of features while preserving essential information, often as a preprocessing step.
- **Anomaly detection:** Identifying unusual or outlier data points.

Important algorithms include K-Means, principal component analysis (PCA), and autoencoders (9).

- **Reinforcement Learning**: The model learns through interaction with an environment.

  Reinforcement Learning (RL) is one of the most exciting and oldest fields in Machine Learning, dating back to the 1950s (9). In reinforcement learning, an intelligent entity called an **agent** learns to make decisions by interacting with its **environment**. The agent observes the current situation, takes actions, and in return receives **rewards** — positive feedback for desirable outcomes and negative feedback (or punishment) for undesirable ones. The agent's goal is to learn the optimal strategy, known as a **policy**, that maximizes its cumulative reward over the long term (9).

  Unlike supervised learning, RL does not rely on labeled training data with correct answers. Instead, the agent learns through trial and error, discovering which actions yield the highest rewards by exploring its environment. This makes RL particularly well-suited for problems involving sequential decision-making, such as game playing (e.g., AlphaGo), robotics, autonomous driving, and stock trading (9).

  A key breakthrough in RL came with the combination of deep learning and reinforcement learning, leading to **deep reinforcement learning**. This approach enabled systems like DeepMind's AlphaGo to defeat world champions in the complex game of Go, and to master a wide range of Atari games using only raw pixel inputs (9). Important RL algorithms include Policy Gradients, Deep Q-Networks (DQN), and Actor-Critic methods (9).

2.2.6 Machine Learning Process

Building a successful Machine Learning model for prediction generally follows seven key steps

1. Gathering Data
2. Data Preparation (80% training & 20% testing)
3. Choose Model
4. Training
5. Evaluation
6. Hyperparameter Tuning
7. Prediction

2.2.7 Limitations of Machine Learning

Despite its effectiveness, Machine Learning has some limitations:

- Requires manual feature engineering
- Difficulty in handling sequential data
- Limited ability to capture long-term dependencies

2.3 Deep Learning

2.3.1 Definition

Deep Learning is a subfield of Machine Learning that uses artificial neural networks with multiple layers to model complex patterns in data (13).

What Is A Neural Network?

Neural networks are among the most remarkable programming concepts ever created. In traditional programming, we tell the computer exactly what operations to perform by breaking down complex problems into many small, simple tasks that it can execute with ease.

In contrast, with a neural network, we do not provide an explicit method to solve the problem. Instead, the computer learns from the provided examples and discovers the appropriate solution on its own.

Artificial neural networks are simple electronic systems inspired by the structure of the human brain.

2.3.2 History

- **1890 :** W. James, a famous American psychologist, introduces the concept of associative memory and proposes what would become a law of functioning for learning in neural networks, later known as Hebb's law.

- **1943 :** McCulloch and Pitts lend their names to a model of the biological neuron (a neuron with binary behavior). They are the first to present the first formal neuron and to show that simple formal neural networks can perform complex logical, arithmetic, and symbolic functions (at least on a theoretical level). They present the first formal neuron.

- **1949 :** D. Hebb, an American physiologist, proposes a learning mechanism and explains conditioning in animals through the properties of the neurons themselves.

- **1958 :** Rosenblatt presents the first artificial neural networks: the perceptron. It is inspired by the visual system and has two layers of neurons ("perceptive" and "decision-making"). He built the first neuro-computer based on this model and applied it to the field of pattern recognition. Note that at that time, the means at his disposal were limited, and it was a technological feat to manage to keep this machine functioning correctly for more than a few minutes.

- **1960 :** B. Widrow, an automation engineer, develops the ADALINE model (Adaptive Linear Element). In its structure, the model resembles the perceptron; however, the learning rule is different. This rule is the origin of the gradient backpropagation algorithm widely used today with multi-layer perceptrons. ADALINE-type networks are still used today for certain specific applications. B. Widrow also founded one of the first companies offering neuro-computers and neuro-components, "Memistor Corporation," at that time.

- **1969 :** M. Minsky and S. Papert publish a work that highlights the theoretical limitations of the perceptron. These limitations were already known, particularly concerning the impossibility of using this model to handle non-linear problems. Limitations of the perceptron; the need for more complex architectures, but how to train them.

- **1967-1982 :** Of course, research did not completely stop. It continued, but in disguise, under the cover of various fields such as adaptive signal processing, pattern recognition, neurobiological modeling, etc. Great names worked during this period, such as S. Grossberg and T. Kohonen.

- **1982 :** J.J. Hopfield is a renowned physicist credited with the resurgence of interest in artificial neural networks. There are several reasons for this: Through

a short, clear, and well-written article, he presents a theory of the functioning and possibilities of neural networks. Note the unconventional presentation of his article. Whereas authors until then had focused on proposing a structure and a learning rule, then studying the emergent properties; J.J. Hopfield first sets the desired behavior for his model and, from there, constructs the structure and learning rule corresponding to the expected result. This model is still widely used today for optimization problems. Furthermore, in the hands of this distinguished physicist, neural network theory became respectable. It was no longer the exclusive domain of certain out-of-touch psychologists and neurobiologists. Finally, a small phrase, placed as a comment in his initial article, highlights the isomorphism of his model with the Ising model (a model of spin glasses). This idea would draw a flood of physicists towards artificial neural networks. Note that at this time, AI was the subject of some disillusionment; it had not met all expectations and had even encountered serious limitations. So, although the perceptron limitations highlighted by Minsky were not lifted by Hopfield's model, research was relaunched.

- **1983 :** The Boltzmann machine is the first known model capable of satisfactorily addressing the limitations noted in the case of the perceptron. However, practical use proved difficult, as the algorithm's convergence was extremely slow (requiring considerable computation time).

- **1986 :** Backpropagation (Rumelhart and McClelland). Rumelhart popularized the gradient backpropagation algorithm, originally conceived by Werbos. Following this discovery, it became possible to achieve a nonlinear input/output function on a network by decomposing this function into a sequence of linearly separable steps. This algorithm allows training the hidden layers of multi-layer networks. New neural network architectures, applications in handwriting recognition, speech recognition and synthesis, vision (image processing).

- **1990 :** Information Society: new applications for web information search and filtering, information extraction, technological monitoring, multimedia (indexing, etc.), data mining, the need to combine several models (28).

Today, neural networks are used in many fields because of their particular properties, notably their learning capability, and because they are dynamic systems.

2.4 From The Biological Neuron To The Artificial Neuron

2.4.1 Analogy with The Brain

The exact functioning of the human brain remains a mystery. However, certain aspects of this astonishing processor are known. In particular, the most fundamental element of the human brain is a specific type of cell which, unlike the rest of the body, does not appear to regenerate. This type of cell is the only part of the body that is not slowly replaced; it is assumed that these cells are the ones that give us the ability to remember, to think, and to apply experiences to each of our actions. These cells, totaling 100 billion, are known as neurons. Each of these neurons can connect with up to 200,000 other neurons, although 1,000 to 10,000 are typical. The power of the human mind comes from the number of these basic components and the multiple connections between them. It also comes from genetic programming and learning. Individual neurons are complicated. They have a myriad of parts, subsystems, and control mechanisms. They convey information via a host of electrochemical pathways. There are more than one hundred different classes of neurons, depending on the classification method used. Together, these neurons and their connections form a process that is not binary, not stable, and not synchronous. Artificial neural networks attempt to reproduce only the most fundamental elements of this complex, versatile, and powerful organism. They do this in a primitive way. But for the software engineer trying to solve problems, neural computing has never been about replicating human brains. It focuses on machines and a new way of solving problems (28).

2.4.2 Definition

A neuron, or nerve cell, is an excitable cell that constitutes the basic functional unit of the nervous system. Neurons ensure the transmission of a bioelectrical signal called a nerve impulse. They have two physiological properties:

- Excitability, i.e., the ability to respond to stimuli and convert them into nerve impulses;
- And conductivity, i.e., the ability to transmit impulses.

2.4.3 Structure of Neurons

A neuron is composed of:

- A cell body, called the **soma**, which is the central part of a neuron containing the cell nucleus. It sums the incoming signals. If this sum exceeds a certain threshold, it itself sends a signal via the axon.

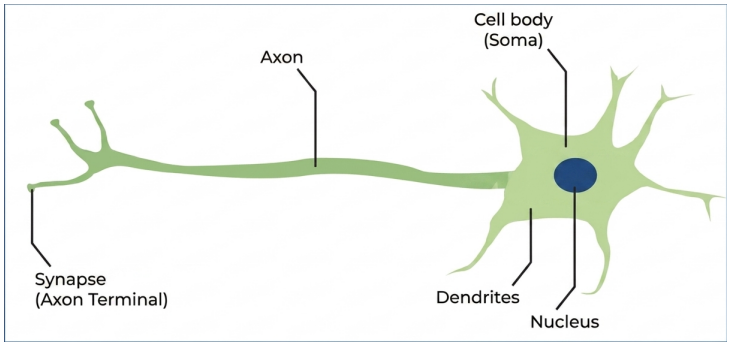


Figure 2.1: Diagram of a Biological Neuron

- A number of fibers called **dendrites**, which are the neuron's main receptors, capturing the signals that arrive at it.
- A long fiber called the **axon**, which transmits signals emitted by the cell body to other neurons.
- **Synapses**, which allow neurons to communicate with each other via axons and dendrites.

Connectionist methods were initiated in the cybernetics era. The researchers' objective was to build a machine capable of reproducing certain aspects of human intelligence as faithfully as possible.

2.5 Artificial Neural Networks (ANN)

2.5.1 Definition

It is based on the hypothesis that intelligent behavior emerges from the structure and behavior of the brain's basic elements that artificial neural networks have developed.

Artificial neural networks are highly interconnected networks of elementary processors (neurons) operating in parallel. Each neuron has an internal state, the activation, through which it influences the other neurons in the network; this activation propagates through the network along synaptic links. The rule that determines the activation of a neuron based on the influence of its peers is called the activation function.

The interest of a neural network is the calculation of a single output value by each output neuron based on the information it receives.

Artificial neural networks are models that can be described by their behaviors, their descriptive variables, and the interactions of their components (27).

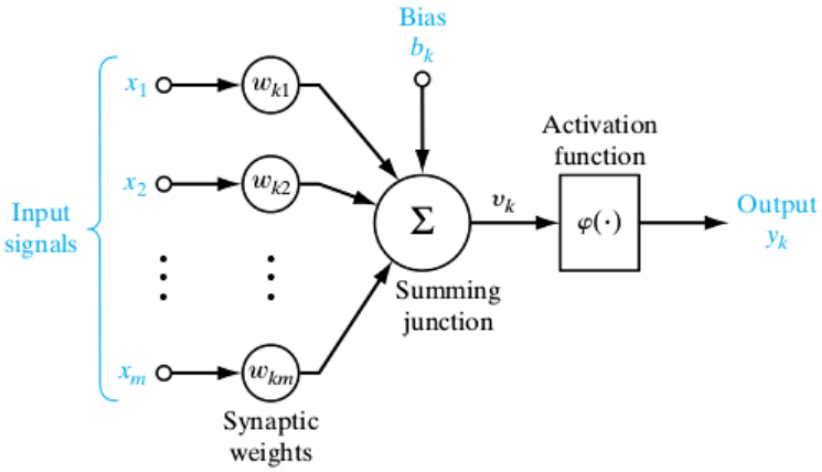


Figure 2.2: Structure of An Artificial Neuron

2.5.2 Artificial Neuron Model

An artificial neuron receives input signals, processes them using weights and bias, and produces an output through an activation function.This transformation is carried out in two steps:

1. The neuron performs a weighted summation of the potentials (superposition principle); the numerical value obtained represents the state of the neuron that emitted it, in order to obtain a global resulting stimulation:

$$Y = \sum w_i x_i \tag{2.1}$$

2. Using a transfer function, the neuron is tested. If this stimulation exceeds a certain threshold, the neuron is activated and transmits a response. In this case:

$$z = f(y) \tag{2.2}$$

Thus the mathematical formula will be of the form:

$$\begin{cases} Y = \sum w_i x_i \\ z = f(y) \end{cases} \tag{2.3}$$

Such that:

- The weighting coefficients w_i, $i = 0, 1, 2, \ldots, n$ are called synaptic weights (w_i is the synaptic weight vector); if w_i is positive, the input x_i is excitatory, whereas if w_i is negative, it is inhibitory(17).

- And x_i, $i = 0, 1, 2, \ldots, n$ are called the inputs (x_i is the input vector).

2.5.3 Behavior

The input vector x_i is multiplied by the synaptic weight vector; this product is summed at the neuron level. The neuron returns only one value.

From this value, an activation function calculates the value of the neuron's state. The output of the neuron is therefore $z = f(y) = f(\sum w_i x_i)$. It is this value that will be transmitted to downstream neurons.

There are many possible forms for the activation function (Figure 2-3). It will be noted that, unlike biological neurons whose state is binary, most transfer functions are continuous, offering an infinite number of possible values within the interval $[0, +1]$ or $[-1, +1]$ depending on the function used (27).

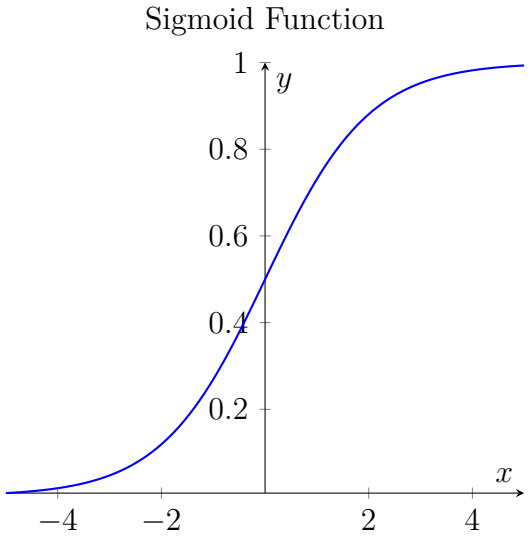
2.5.4 Activation Function

An activation function is a mathematical function commonly used in machine learning, statistics, and neural networks. It play a crucial role in determining the output of neural network layers. They introduce non-linearity into the network, allowing it to learn and perform complex tasks and decide whether a neuron should fire(Activate)or not.

Common activation functions include:

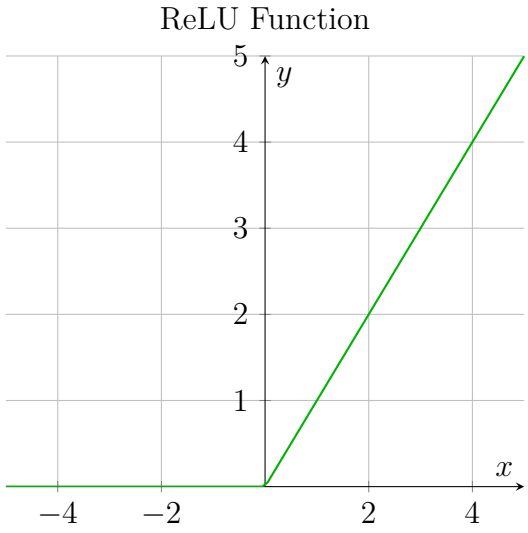
- **sigmoid:** utilized for binary classification (Cat vs. Not-Cat, for example). Values between 0 and 1, which can be thought of as a probability, are squashed.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



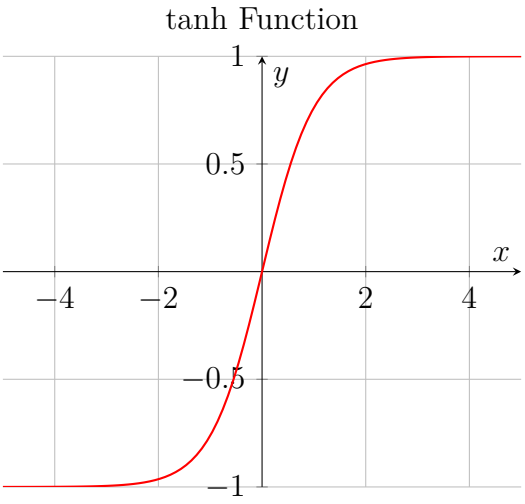
- **ReLU(Rectified Linear Unit):**The industry standard of hidden layers. Training is accelerated by its quick computation and the ability to avoid the "vanishing gradient" issue.

$$\text{ReLU}(x) = \max(0, x)$$



- **tanh(Hyperbolic Tangent):**Because it is zero-centered (output ranges from -1 to 1), which keeps the model's math balanced during optimization, it is superior to Sigmoid for hidden layers.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



2.5.5 Network Architecture

The network architecture describes the structure of the neural network, including its layers and the flow of information between them.

The network consists of:

- an input layer, which receives the time series data.
- one or more hidden layers, where the network learns patterns from the data.
- an output layer, which produces the predicted value.

Each layer has a specific role in transforming the input data into meaningful output. The hidden layers extract important features and relationships from the time series, enabling the network to learn underlying patterns such as trends and seasonality.

This architecture allows the model to improve forecasting performance by learning from past observations."

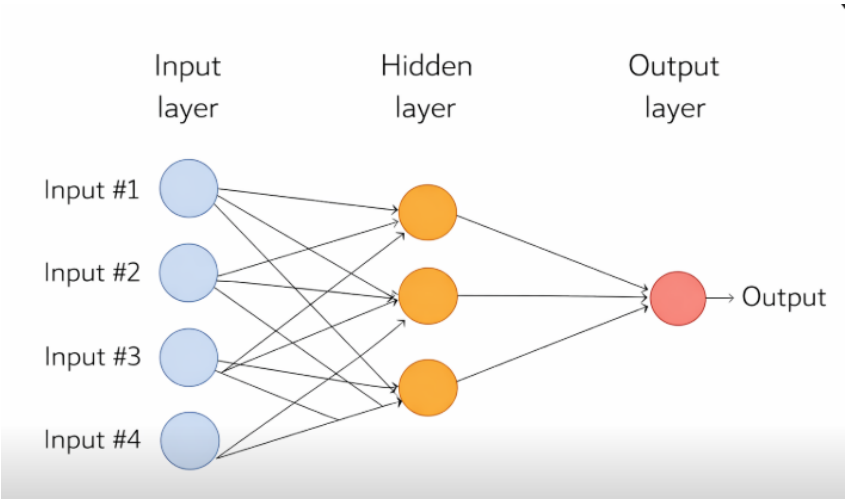


Figure 2.3: Basic Artificial neural network.

2.5.6 Advantages of Deep Learning

Deep Learning offers several advantages:

- Automatic feature extraction
- Ability to model nonlinear relationships
- High performance with large datasets

2.5.7 Machine Learning vs Deep Learning

Machine Learning	Deep Learning
Requires feature engineering	Automatic feature extraction
Works with smaller datasets	Requires large datasets
Uses simple models	Uses neural networks

Table 2.1: Comparison between Machine Learning and Deep Learning

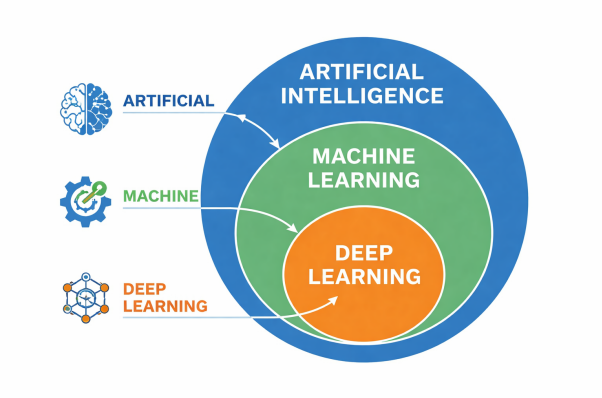


Figure 2.4: AI Subfields

2.6 Recurrent Neural Networks

2.6.1 Definition

A Recurrent Neural Network (RNN) is a class of artificial neural networks within Machine Learning specifically designed to model and analyze sequential data. Unlike feedforward neural networks, an RNN incorporates recurrent connections that allow information to persist across time steps through an internal hidden state. This structure enables the network to capture temporal dependencies by using both current inputs and previously stored information when producing outputs. RNNs are particularly well-suited for tasks involving ordered data, such as time series forecasting, natural language processing, and speech recognition.

2.6.2 Applications of RNN

RNNs are widely used in:

- Time series forecasting.
- Speech recognition.
- Natural language processing.
- Handwriting recognition.
- Video activity recognition.

2.6.3 Architecture of RNN

A recurrent neural network consists of interconnected units (neurons) interacting non-linearly, for which there exists at least one cycle in the structure. The units are connected by arcs (synapses) that have weights. The output of a neuron is a non-linear combination of its inputs.

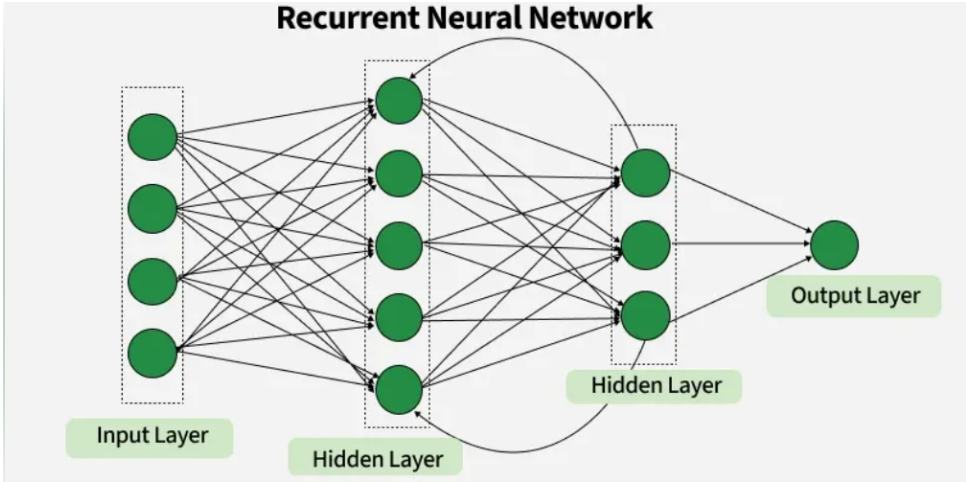


Figure 2.5: RNN Architecture

2.6.4 Difference Between ANN and RNN

Unlike a standard Artificial Neural Network (ANN), where data flows strictly in one direction from input to output without forming cycles, a Recurrent Neural Network (RNN) incorporates recurrent connections that create feedback loops (5). This fundamental architectural difference gives the RNN a form of internal memory: it maintains a "hidden state" that retains information from previous inputs in a sequence, allowing the network to use past context when processing the current input (6). Consequently, while FNNs are well-suited for static data like individual images or tabular records, RNNs excel at tasks where order and temporal dependencies matter, such as time series forecasting, natural language processing, and speech recognition (5; 7). However, this recurrent structure introduces significant training challenges, most notably the vanishing and exploding gradient problems, which can prevent the network from learning long-range dependencies in a sequence (7).

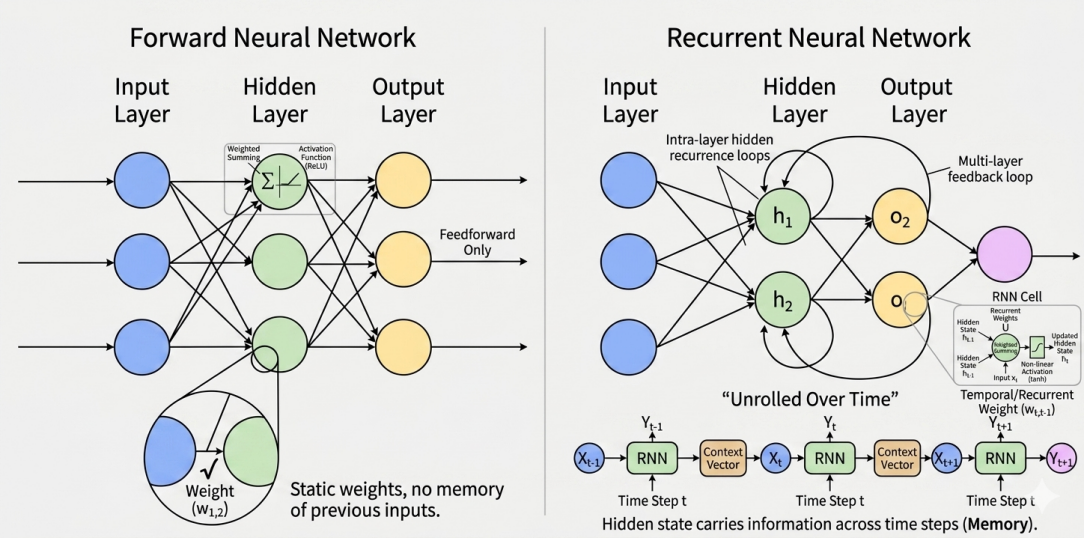


Figure 2.6: Difference Between ANN and RNN

2.6.5 Working Principle

The working principle of a recurrent neural network relies on the concept of a **hidden state** that acts as a memory of past information. At each time step t, the network processes two inputs: the current input x_t and the previous hidden state h_{t-1}. The new hidden state h_t is computed as:

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{2.4}$$

where:

- W_{xh} is the weight matrix connecting the input to the hidden layer
- W_{hh} is the recurrent weight matrix connecting the hidden layer to itself across time steps
- b_h is the bias term
- f is a non-linear activation function (typically tanh or ReLU)

The output y_t at each time step is then computed from the hidden state:

$$y_t = g(W_{hy}h_t + b_y) \tag{2.5}$$

where g is an activation function (e.g., softmax for classification tasks).

The same weights (W_{xh}, W_{hh}, W_{hy}) are shared across all time steps, which allows the network to process sequences of arbitrary length and learn temporal dependencies. When "unrolled" in time, the RNN becomes equivalent to a very deep feedforward network with tied weights.

2.6.6 Types of RNN

Here are five types of RNN:

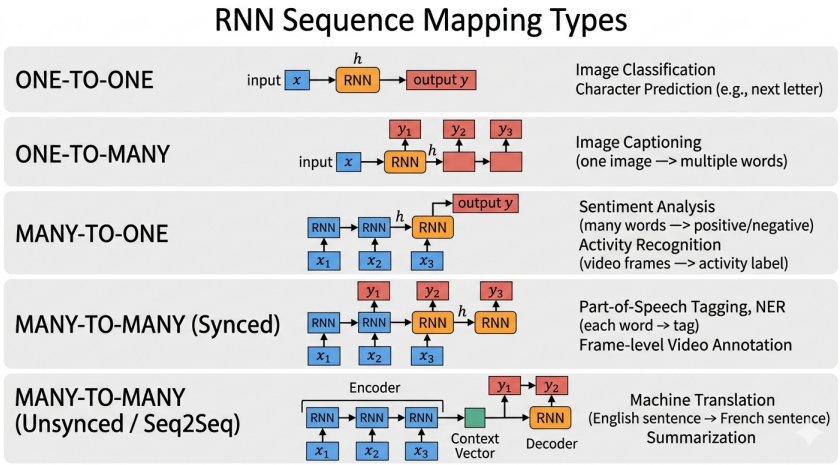


Figure 2.7: Types of Recurrent Neural Networks (RNNs)

Examples:

- **one to one :** Classifying whether an email is spam or not spam.
  How it works : A single email (input) is processed to produce a single classification label (output: spam or not spam).

- **one to many :** Generating a caption for an image.
  How it works : A single image is fed into the network, which generates a sequence of words describing the image.

- **many to one :** Stock price prediction
  How it works : using a sequence of past stock prices to predict tomorrow's closing price.

- **many to many (Same Length) :** Part-of-Speech (POS) tagging – assigning a grammatical tag to each word in a sentence.
  How it works : Each word in the input sentence receives a corresponding POS tag at the same time step.

- **many to many (Different Length / Sequence-to-Sequence) :** Translations can be a good example of a many-to-many type of network.
  How it works : The entire input sequence is encoded into a context vector, then decoded into an output sequence which may have a different length.

2.6.7 Gradient

In machine learning, a gradient is a vector of partial derivatives that points in the direction of the steepest increase of a loss function. It acts as a mathematical compass that tells an AI model exactly how to alter its parameters (weights and biases) to reduce errors and learn from data.

The Core Concept

When a machine learning model makes predictions, a loss function measures how wrong those predictions are. The goal of training is to minimize this loss.

Because a model can have millions of parameters, you cannot just guess the best values. The gradient solves this by calculating the derivative of the loss function with respect to every single weight in the network:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

- **The Gradient Vector:** Contains the individual slope for every parameter.
- **The Direction:** Points toward the fastest increase in error (uphill).
- **The Magnitude:** Indicates how steep the slope is. A larger value means a bigger error correction is needed.

Gradient Descent (How Models Learn)

To improve the model, an optimization algorithm called Gradient Descent moves the parameters in the opposite direction of the gradient (downhill).

The fundamental update rule for a parameter w is:

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \frac{\partial L}{\partial w_{\text{old}}}$$

- **The Minus Sign ($-$):** Forces the model to go down the hill toward minimum error.
- **The Learning Rate (α):** A small multiplier (e.g., 0.01) that controls how big of a step the model takes.

Example of the Calculation of the Gradient

To understand how gradients work in practice, let's look at a concrete example of training a single neuron (a neural network with one layer and one output) to learn the **AND** logic gate. We want the network to take two inputs (x_1, x_2) and predict the correct output (y).

Step 1: The Setup (Data & Network)

- **Training Data:**
  - Input: $\mathbf{x} = [1, 1]$, Target: $y = 1$
- **Initial Parameters:**
  - Weights: $w_1 = 0.5$, $w_2 = 0.5$
  - Bias: $b = -0.7$
- **Learning Rate (η):** 0.1

Step 2: Step-by-Step Training Iteration

Step A: Forward Pass (Prediction) First, calculate the weighted sum of inputs (z):

$$z = (x_1 \cdot w_1) + (x_2 \cdot w_2) + b$$

$$z = (1 \cdot 0.5) + (1 \cdot 0.5) - 0.7 = 0.3$$

Next, pass z through a Sigmoid Activation Function to get the prediction $(\hat{y})$:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-0.3}} \approx 0.574$$

Step B: Compute the Loss (Error) We use the Mean Squared Error (MSE) loss function for this single sample:

$$L = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(1 - 0.574)^2 \approx 0.091$$

Step C: Backward Pass (Calculating Gradients) To know how to change the weights, we find the gradient of the loss using the Chain Rule:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

1. Loss derivative: $\frac{\partial L}{\partial \hat{y}} = -(y - \hat{y}) = -(1 - 0.574) = -0.426$

2. Sigmoid derivative: $\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y}) = 0.574 \cdot (1 - 0.574) \approx 0.245$

3. Input derivative: $\frac{\partial z}{\partial w_1} = x_1 = 1$

Multiply them together to get the gradient for w_1:

$$\nabla L(w_1) = -0.426 \cdot 0.245 \cdot 1 \approx -0.104$$

(Because $x_2 = 1$, the gradient for w_2 is also -0.104. The gradient for the bias b is $-0.426 \cdot 0.245 \cdot 1 = -0.104$)

Step D: Update the Parameters We update the parameters by subtracting a fraction of the gradient:

$$w_{\text{new}} = w_{\text{old}} - (\eta \cdot \nabla L)$$

- New w_1: $0.5 - (0.1 \cdot -0.104) = \mathbf{0.5104}$
- New w_2: $0.5 - (0.1 \cdot -0.104) = \mathbf{0.5104}$
- New b: $-0.7 - (0.1 \cdot -0.104) = \mathbf{-0.6896}$

Step 3: The Result

If we run the forward pass again with the new weights:

- New $z = (1 \cdot 0.5104) + (1 \cdot 0.5104) - 0.6896 = 0.3312$
- New $\hat{y} = \sigma(0.3312) \approx \mathbf{0.582}$

The prediction moved from 0.574 closer to our target of 1. The network successfully learned a little bit from its mistake.

Remarks on Gradient Interpretation

The gradient for w_1 (written as $\frac{\partial L}{\partial w_1}$) is a number that tells you how much the total error (loss) changes if you make a tiny adjustment to the weight w_1. It represents the sensitivity and direction of the model's error relative to that specific weight.

What the Value Means

When you calculate this gradient during training, the resulting number gives the optimization algorithm two critical pieces of information:

- **The Sign (+ or −)** indicates the direction of the error:

  - **Positive Gradient (+):** Increasing w_1 will increase the error (you should decrease w_1).
  - **Negative Gradient (−):** Increasing w_1 will decrease the error (you should increase w_1).

- **The Magnitude (Size)** indicates the steepness:
  - A large number means a small change to w_1 causes a massive drop or spike in error.
  - A number close to zero means changing w_1 barely affects the error anymore.

Vanishing and Exploding Gradients

To illustrate the **vanishing gradient** and the **exploding gradient** phenomena using our example, we need to imagine that our network no longer has just a single layer, but several successive layers instead. The Chain Rule multiplies the derivatives of each layer together. If we add multiple layers, we end up multiplying many values with one another.

Example of a Vanishing Gradient

This phenomenon occurs when gradients become progressively smaller during backpropagation until they essentially shrink to zero. As a result, the earliest layers of the network stop learning entirely.

Scenario in a Deep Network (e.g., 4 Layers) If we chain 4 neurons of this same type together, the gradient calculation for the very first weight w_1 will multiply the Sigmoid derivatives of each individual layer:

$$\frac{\partial L}{\partial w_1} = \text{Loss Derivative} \times \sigma'(z_4) \times \sigma'(z_3) \times \sigma'(z_2) \times \sigma'(z_1) \times x_1$$

If each Sigmoid layer has an average derivative value of 0.2, their product becomes:

$$0.2 \times 0.2 \times 0.2 \times 0.2 = \mathbf{0.0016}$$

- **The Quantitative Result:**

$$\nabla L(w_1) \approx -0.426 \times 0.0016 \times 1 = \mathbf{-0.00068}$$

- **Consequence:** During the parameter update step:

$$w_{\text{new}} = 0.5 - (0.1 \times -0.00068) = \mathbf{0.500068}$$

The weight barely changes, meaning weight evolution grinds to a halt and the network gets stuck.

Example of an Exploding Gradient

This phenomenon is the exact inverse: gradients accumulate increasingly large values, forcing the weights to oscillate wildly or completely diverge until they become `NaN` values.

The Cause in Our Example: Excessively Large Initial Weights Imagine that we initialized our weights not at 0.5, but at much larger values—for instance, $w = 5.0$ at every single layer. Additionally, assume we are using a linear activation function (without the squeezing effect of the Sigmoid) where the derivative with respect to the activation is equal to the weight itself.

Scenario in a Deep Network (e.g., 4 Layers) During backpropagation, the Chain Rule multiplies the layer weights together to pass the error signal backward through the network architecture:

$$\frac{\partial L}{\partial w_1} = \text{Loss Derivative} \times w_4 \times w_3 \times w_2 \times x_1$$

If all of our weights are equal to 5.0, this geometric product explodes:

$$5 \times 5 \times 5 = \mathbf{125}$$

- **The Quantitative Result:**

$$\nabla L(w_1) \approx -0.426 \times 125 = \mathbf{-53.25}$$

- **Consequence:** Performing the parameter update step with our learning rate $\eta = 0.1$:

$$w_{\text{new}} = 5.0 - (0.1 \times -53.25) = \mathbf{10.325}$$

The weight more than doubles in a single training iteration. In the very next iteration, the gradient will be multiplied by $10.325^3 \approx 1100$, driving the weight to an astronomical value. The model instantly diverges.

2.6.8 RNN problems

Standard Recurrent Neural Networks are notoriously difficult to train on long sequences due to two interrelated issues: the vanishing gradient problem and the exploding gradient

problem. Both arise from the repeated multiplication of the gradient by the recurrent weight matrix during backpropagation through time (BPTT).

- **Vanishing Gradient:** As the network goes through numerous stages during training over lengthy periods, the gradients that aid in learning may get smaller. As a result, the model finds it challenging to learn long-term patterns because prior knowledge becomes almost meaningless (8).
- **Exploding Gradient:**When gradients get too big, they might become unstable during training. This causes inconsistent updates, which hinders the model's ability to learn effectively (8).
- **Impact on RNNs:** Standard RNNs' capacity to capture long-term dependencies is limited by both vanishing and exploding gradients, which make it difficult for them to recall and apply data from previous time steps (8).

The vanishing gradient problem causes the network to ignore early inputs, limiting its memory to recent time steps. The exploding gradient problem makes training unstable, often producing numerical overflow.

2.6.9 Limitations of RNNs for Forecasting

While RNNs are theoretically well-suited for time series forecasting, their practical application faces several specific challenges:

- **Short-term memory**: Due to the vanishing gradient problem, standard RNNs struggle to capture long-range dependencies beyond approximately 10-20 time steps. This makes them unsuitable for forecasting tasks with seasonal patterns or long-term trends.
- **Sensitivity to sequence length**: The performance of RNNs degrades as the input sequence length increases, limiting their ability to process long historical datasets common in forecasting.
- **Instability during training**: The exploding gradient problem can cause numerical overflow, leading to unreliable model convergence and inconsistent forecast accuracy.
- **Inability to capture multiple seasonal patterns**: Standard RNNs lack the gating mechanisms required to handle complex temporal patterns such as daily, weekly, and yearly cycles simultaneously.

These limitations have motivated the development of more advanced architectures, such as LSTM network, which is discussed in Chapter 3.

Chapter 3

Forecasting Time Series with LSTM

As established in the previous chapter, Deep Learning (DL) provides a powerful framework for learning hierarchical representations from data. Among its core architectures, Recurrent Neural Networks (RNNs) were specifically designed to handle sequential information by maintaining a hidden state that captures dependencies across time steps. This made RNNs the de facto choice for tasks such as time series prediction, language modeling, and speech recognition.

However, RNNs suffer from a critical limitation: the difficulty of learning long-range dependencies due to the vanishing and exploding gradient problems. In practice, standard RNNs struggle to retain information from inputs that are more than a handful of steps apart. To overcome this fundamental bottleneck, this chapter introduces a specialized variant of recurrent architectures: the Long Short-Term Memory (LSTM) network. The LSTM cell achieves this through its core gating mechanisms—namely the forget gate, input gate, and output gate—which regulate the flow of information into a separate cell state, allowing the network to preserve or discard information over arbitrarily long time intervals.

3.1 Definition

Long Short-Term Memory (LSTM) networks are a class of neural architectures designed for sequential data, such as speech, text, or time series (8). Unlike traditional RNNs, which suffer from vanishing or exploding gradients (18), LSTMs employ gating mechanisms to store, update, and retrieve information, enabling them to capture long-term dependencies more effectively. The key characteristics of an LSTM network are as follows:

- captures long-term dependencies by storing data over time steps.
- controls the flow of information using input, forget, and output gates.

- effectively manages lengthy sequences without experiencing vanishing gradient problems.

3.2 Applications of LSTM

LSTM networks are widely used in:

- Stock price prediction
- Weather forecasting
- Speech recognition

3.3 The Architecture of LSTM

The LSTM model is based on a specialized architecture that enables it to effectively capture long-term dependencies in sequential data. The figure below presents the structure of an LSTM cell and its key components.

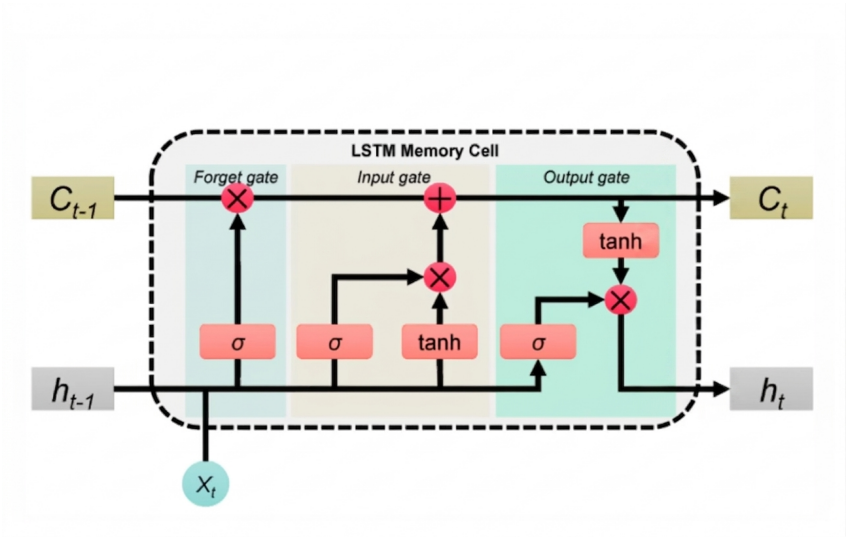


Figure 3.1: LSTM Architecture

3.3.1 LSTM Gates

Forget Gate (Selective Erasure)

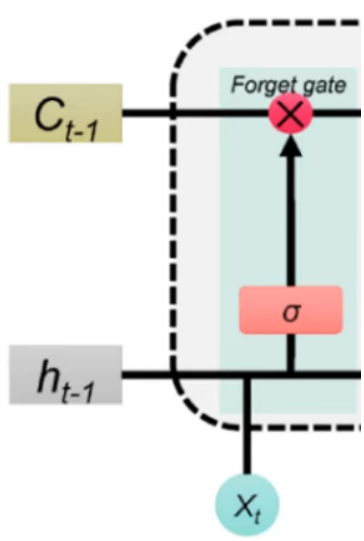


Figure 3.2: Forget Gate Architecture

The forget gate determines what information from the previous cell state c_{t-1} should be retained or discarded. It enables the LSTM to selectively reset portions of its memory when they are no longer relevant to the task. (10) identified the forget gate as a critical addition to the original LSTM architecture, allowing the network to learn when to clear its memory.

Mathematical Definition

$$f_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_f) \tag{3.1}$$

where:

- $\sigma(\cdot)$ is the logistic sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$, outputting in $(0, 1)$,
- $\mathbf{W}_f \in \mathbb{R}^{H\times(H+D)}$ is the forget gate weight matrix,
- $\mathbf{b}_f \in \mathbb{R}^H$ is the forget gate bias.

Symbol	Meaning
H	Number of LSTM units (size of the hidden state / memory)
D	Dimension of the input data
$H \times (H + D)$	Size of the forget gate weight matrix
$[\ldots;\ldots]$	Concatenation.

Interpretation and Behavior

For each memory cell dimension $j \in \{1, \ldots, H\}$:

- $f_{t,j} \approx 1$ **(gate open):** The corresponding value in $c_{t-1,j}$ is fully retained.

- $f_{t,j} \approx 0$ **(gate closed):** The corresponding value is completely erased (set to 0).
- **Intermediate values:** Partial retention.

(10) made a crucial observation regarding initialization: setting the forget gate bias $\mathbf{b}_f$ to a positive value (e.g., 1.0 or 2.0) encourages the network to initially keep memory open, preventing rapid forgetting early in training. This recommendation has become standard practice in LSTM implementations (13).

Role in Memory Management

The forget gate provides the LSTM with the ability to perform several critical operations:

1. **Reset irrelevant memory:** After a sentence boundary, the network can forget the previous subject's grammatical number or gender.
2. **Clear outdated information:** When a new context is established, old contextual information can be discarded.
3. **Prevent memory saturation:** Without forgetting, cell states could grow unbounded or become saturated, reducing representational capacity.

Input Gate (Controlled Writing)

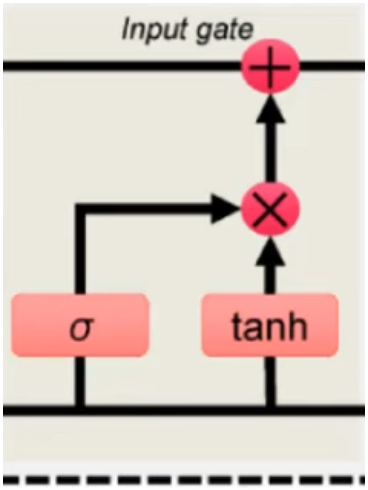


Figure 3.3: Input Gate Architecture

The input gate determines how much of the candidate memory $\tilde{c}_t$ should be written into the cell state. This mechanism prevents the cell state from being overwritten by every input, allowing the network to selectively update its memory only when relevant new information arrives (18).

Mathematical Definition

$$i_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_i) \tag{3.2}$$

where:

- $\mathbf{W}_i \in \mathbb{R}^{H \times (H+D)}$ is the input gate weight matrix,
- $\mathbf{b}_i \in \mathbb{R}^H$ is the input gate bias.

Interpretation

For each memory cell dimension:

- $i_t \approx 1$ **(gate open):** The candidate value $\tilde{c}_t$ is fully written into the cell state.
- $i_t \approx 0$ **(gate closed):** The cell state remains unchanged from c_{t-1} (except for any forgetting applied via f_t).

Interaction with The Forget Gate

The input and forget gates work in tandem to enable four distinct memory operations, as summarized in Table 3.3.1.

f_t (Forget)	i_t (Input)	Effect
≈ 1 (retain)	≈ 0 (don't write)	**Preserve** existing memory
≈ 1 (retain)	≈ 1 (write)	**Update** memory by adding new info
≈ 0 (forget)	≈ 0 (don't write)	**Clear** memory (set near zero)
≈ 0 (forget)	≈ 1 (write)	**Replace** memory with new info

Table 3.1: Memory operations enabled by forget and input gate interactions

Why Separate Input and Candidate?: The separation of i_t (sigmoid gate) and $\tilde{c}_t$ (tanh candidate) is deliberate. The sigmoid gate decides whether to store, while the tanh candidate decides what to store (including sign/direction). Without this separation, the same activation would need to simultaneously encode both the storage decision and the content, substantially reducing representational power (16).

Output Gate (Controlled Reading):

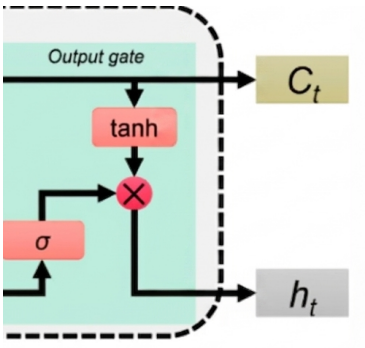


Figure 3.4: Output Gate Architecture

The output gate controls which parts of the cell state c_t are exposed as the hidden state $\mathbf{h}_t$. This separation allows the LSTM to maintain information in the cell state without necessarily outputting it at every time step, enabling the network to retain memory while focusing on other aspects of the input (18).

Mathematical Definition

$$o_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_o) \tag{3.3}$$

$$\mathbf{h}_t = o_t \odot \tanh(c_t) \tag{3.4}$$

where:

- $\mathbf{W}_o \in \mathbb{R}^{H\times(H+D)}$ is the output gate weight matrix,
- $\mathbf{b}_o \in \mathbb{R}^H$ is the output gate bias,
- $\tanh(c_t)$ squashes the cell state to $(-1, 1)$ before gating.
- $\odot$ represents element-wise multiplication.

Interpretation

For each memory cell dimension:

- $o_t \approx 1$ **(gate open):** The normalized cell state value is fully propagated to the hidden state.
- $o_t \approx 0$ **(gate closed):** The hidden state receives no information from this memory cell.

The Role of tanh on Cell State Output

Applying tanh to c_t before output serves two important purposes:

1. **Normalization:** Squashes the cell state to $(-1, 1)$, preventing the hidden state from having excessively large magnitude, which would make training unstable.
2. **Nonlinearity:** Introduces a nonlinear transformation while preserving the sign information of the cell state (the tanh function is odd: $\tanh(-z) = -\tanh(z)$).

Separation of Memory and Output

The output gate enables a crucial capability: **retaining information in memory while outputting something else**. (19) provide concrete examples:

> In a language model processing "The cat... (long clause) ...runs":
>
> - The cell state tracks that the subject is singular.
> - During intervening words, the output gate may hide this information (not needed at those positions).
> - When reaching the verb, the output gate opens to expose the singular number for verb agreement prediction.

This decoupling of memory storage from output generation is impossible in standard RNNs, where memory and output are the same state (20).

3.3.2 Cell state (Core Memory Unit)

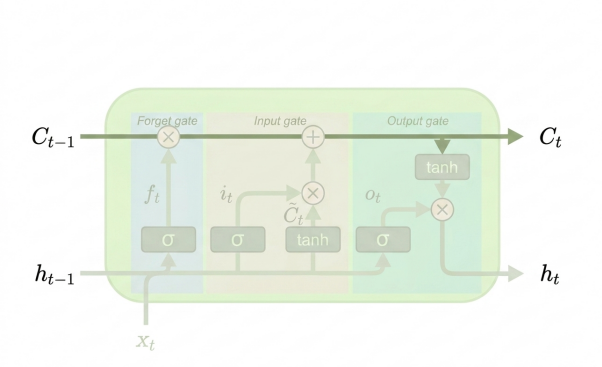


Figure 3.5: Cell State Architecture

The cell state, denoted $c_t \in \mathbb{R}^H$ at time step t (where H is the number of memory cells), serves as the LSTM's long-term memory register. Unlike the hidden state in a vanilla RNN, which is completely rewritten at every step through a nonlinear transformation,

the cell state flows through the network with only **linear, additive updates**.
The cell state c_t is a vector that persists across time steps and is modified only through controlled additive operations. It provides a protected pathway for information to travel over long sequences without undergoing repeated nonlinear transformations.

Mathematical Formulation

The fundamental cell state update equation is:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{3.5}$$

where:

- $c_{t-1} \in \mathbb{R}^H$ is the previous cell state,
- $f_t \in [0,1]^H$ is the forget gate activation,
- $i_t \in [0,1]^H$ is the input gate activation,
- $\tilde{c}_t \in [-1,1]^H$ is the candidate memory value,
- $\odot$ denotes element-wise (Hadamard) multiplication.

Additive Updates: The additive nature of Equation 3.5 is crucial. New information enters via addition ($+i_t \odot \tilde{c}_t$) rather than multiplication, preventing the multiplicative accumulation of transformations that causes gradient vanishing in standard RNNs.

Gradient Flow Analysis

The additive update mechanism directly addresses the vanishing gradient problem. Consider the gradient of the loss $\mathcal{L}$ with respect to the cell state at time $t-1$:

$$\frac{\partial \mathcal{L}}{\partial c_{t-1}} = \frac{\partial \mathcal{L}}{\partial c_t} \cdot \frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial \mathcal{L}}{\partial c_t} \cdot f_t \tag{3.6}$$

Because this is element-wise multiplication by f_t (rather than multiplication by a full weight matrix), we have:

$$\left\| \frac{\partial \mathcal{L}}{\partial c_{t-1}} \right\| = \left\| f_t \odot \frac{\partial \mathcal{L}}{\partial c_t} \right\| = |f_t| \cdot \left\| \frac{\partial \mathcal{L}}{\partial c_t} \right\| \tag{3.7}$$

When the forget gate is open ($f_t \approx 1$), the gradient magnitude is preserved across the time step. When backpropagating through T steps:

$$\frac{\partial \mathcal{L}}{\partial c_{t-T}} = \left(\prod_{k=1}^{T} f_{t-k+1} \right) \odot \frac{\partial \mathcal{L}}{\partial c_t} \tag{3.8}$$

If the forget gates remain near 1 for long periods—a behavior that networks empirically learn—the gradient does not vanish exponentially. This stands in stark contrast to standard RNNs, where gradients decay as $(\mathbf{W}_{hh}^T)$ (24).

Candidate Memory Generation

We define the candidate memory $\tilde{c}_t$, which proposes new information to potentially store in the cell state:

$$\tilde{c}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_c) \tag{3.9}$$

where:

- $\mathbf{W}_c \in \mathbb{R}^{H \times (H+D)}$ is the weight matrix (D is the input dimension),
- $\mathbf{h}_{t-1} \in \mathbb{R}^H$ is the previous hidden state,
- $\mathbf{x}_t \in \mathbb{R}^D$ is the current input,
- $[\mathbf{h}_{t-1}; \mathbf{x}_t]$ denotes vector concatenation,
- $\mathbf{b}_c \in \mathbb{R}^H$ is the bias term.

The tanh activation squashes the output to $[-1, 1]$, ensuring candidate values have zero mean. This property stabilizes learning by preventing bias drift across time steps (14).

These gates regulate the flow of information and allow the network to selectively remember or forget information (Sood).

These equations describe how information flows through the network and how long-term dependencies are preserved.

3.4 Complete LSTM Cell Formulation

3.4.1 Combined Equations

For completeness, the full LSTM forward pass is:

$$f_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_f) \tag{3.10}$$

$$i_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_i) \tag{3.11}$$

$$o_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_o) \tag{3.12}$$

$$\tilde{c}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_c) \tag{3.13}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{3.14}$$

$$\mathbf{h}_t = o_t \odot \tanh(c_t) \tag{3.15}$$

3.4.2 Parameter Count

A standard LSTM layer contains:

- 4 weight matrices: $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_o, \mathbf{W}_c$, each of size $H \times (H + D)$
- 4 bias vectors: $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_o, \mathbf{b}_c$, each of length H

The total number of trainable parameters is:

$$\text{Params} = 4H(H + D + 1) \tag{3.16}$$

where the $+1$ accounts for the bias terms. This is approximately four times the parameter count of a standard RNN with the same hidden dimension (12).

3.4.3 Computational Graph Description

The information flow through an LSTM cell can be described as follows:

1. The previous hidden state $\mathbf{h}_{t-1}$ and current input $\mathbf{x}_t$ are concatenated.
2. This concatenated vector is passed through four separate affine transformations followed by activations:
   - Sigmoid for f_t, i_t, o_t (three independent gates)
   - tanh for $\tilde{c}_t$ (candidate memory)
3. The previous cell state c_{t-1} is multiplied by f_t (selective forgetting).
4. The candidate $\tilde{c}_t$ is multiplied by i_t (selective writing).
5. These two results are added to produce the new cell state c_t.
6. The new cell state is squashed via tanh and multiplied by o_t to produce $\mathbf{h}_t$.

3.4.4 Example: Character-Level Language Modeling

For a network trained to predict the next character in text sequences, (19) observed the following specialized behaviors:

- **Cell state neurons** learned to count parentheses depth, track whether the model is inside quotes, remember that the next character should be capitalized, and maintain line position in structured text.
- **Forget gate** learned to reset these counters precisely when parentheses close, quotes end, or line breaks occur.
- **Input gate** learned to update counters exactly when opening parentheses or quotes are encountered.
- **Output gate** learned to expose the state only when predicting characters that depend on context (e.g., only closing a quote if the quote was previously opened).

(16) conducted a comprehensive empirical comparison of eight LSTM variants, concluding that while certain modifications (e.g. peephole connections) can provide marginal benefits, the standard architecture with forget gates remains highly effective across a wide range of tasks.

3.5 The Working Mechanism of LSTM:

The LSTM processes input data step-by-step, updating its memory cell, and producing outputs based on both current input and past information. This mechanism allows LSTM networks to capture long-term dependencies in sequential data (23).

The working mechanism of LSTM is based on gated units, including the input, forget, and output gates, which regulate the flow of information through the cell state.

Methodology

The methodology consists of data preprocessing, training an LSTM model, and evaluating forecast performance.

Figure 3.6: General Methodology Flowchart

3.6 Step By Step Implementation

Here, we use the Keras library, which offers a high-level interface for creating and training neural networks, to develop LSTM networks in R.

Step 1: Install And Load Required Libraries

Begin by installing the necessary packages, followed by loading the Keras and TensorFlow libraries. These powerful libraries enable you to build and train deep learning architectures directly within R.

```
install.packages("keras")
install.packages("tensorflow")

library(keras) # a high-level API for building neural networks
   easily
library(tensorflow) #the backend engine that does the heavy
   mathematical calculations
```

Step 2: Prepare The Training Dataset

Generate a sample dataset to train the LSTM model. Structure the input data as a 3D array with dimensions for samples, timesteps, and features, then convert the labels into categorical form through one-hot encoding.

```
 X <- array(0, dim = c(n_samples, timesteps, features))
y <- array(0, dim = c(n_samples))
```

such that :

- Samples = How many examples
- Timesteps = How many steps back to look
- Features = What kind of measurements (price, temperature, etc.)

Step 3: Define The LSTM Model

Using the Sequential API, we construct the neural network with an LSTM layer and a final Dense output layer employing sigmoid activation.

```
   model <- keras_model_sequential()

# Add LSTM layer
model %>%
  layer_lstm(units = 50,
             input_shape = c(timesteps, features),
             return_sequences = FALSE) %>%

  # Add Dense output layer with softmax activation for
     classification
  layer_dense(units = 3, activation = "sigmoid")
```

Step 4: Compile And Inspect The Model

Before starting the training process, first compile the model — specify the loss function, choose an optimizer, and set the evaluation metric. After compilation, display the model's architecture.

```
   model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adam(learning_rate = 0.001),
  metrics = c("accuracy")
)

# Display the model architecture
summary(model)
```

Step 5: Train The LSTM Model

Use the training dataset to train the model, specifying three key parameters: the number of epochs, the batch size, and the validation split fraction.

```
    history <- model %>% fit(
  x = X,
  y = y_categorical,
  epochs = 50,
```

```
  batch_size = 32,
  validation_split = 0.2,
  verbose = 1
)
```

Step 6: Evaluate The Model

Upon completing the training process, we test the model's performance against the dataset to obtain the loss and accuracy values of the trained LSTM model.

Scale-Dependent Metrics

$$\text{MAE} = \frac{1}{n}\sum_{t=1}^{n}|y_t - \hat{y}_t| \tag{3.17}$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{y}_t)^2} \tag{3.18}$$

$$\text{MAPE} = \frac{100\%}{n}\sum_{t=1}^{n}\left|\frac{y_t - \hat{y}_t}{y_t}\right| \tag{3.19}$$

```
    evaluation <- model %>% evaluate(X, y_categorical, verbose =
       0)
cat("\nEvaluation␣Results:\n")
cat(paste("Loss:", round(evaluation["loss"], 4), "\n"))
cat(paste("Accuracy:", round(evaluation["accuracy"], 4), "\n"))
```

Step 7: Generate Predictions

We apply the trained model to make predictions on the input data. The resulting output displays the predicted probability for every class.

```
  predictions <- model %>% predict(X[1:5, , , drop = FALSE],
     verbose = 0)
```

3.7 LSTM for Time Series Forecasting

LSTM networks are particularly well-suited for time series forecasting due to their ability to learn temporal dependencies. Unlike traditional models, LSTM can model complex nonlinear relationships and capture long-term patterns in the data (23) .

3.8 Application on Time Series Datasets

The two next scripts and figures present forecasting results from a Long Short-Term Memory (LSTM) network applied to two different time series datasets:

1. **AirPassengers**: monthly totals of airline passengers (a classic seasonal time series with an upward trend).
2. **co2**: monthly measurements of atmospheric carbon dioxide (co_2) concentration, also seasonal with a rising trend.

For AirPassengers Dataset

```
    # Load required libraries
library(keras) #  for building neural networks
library(tensorflow) # which acts as the "engine" behind Keras
library(ggplot2) # for advanced graphics

library(dplyr) # for easy data manipulation
library(tidyr) # for cleaning and reshaping data

# --- Load Data ---
data <- c(
  145, 150, 178, 163, 172, 178, 199, 199, 184, 162, 146, 166,
  171, 180, 193, 181, 183, 218, 230, 242, 209, 191, 172, 194,
  196, 196, 236, 235, 229, 243, 264, 272, 237, 211, 180, 201,
  204, 188, 235, 227, 234, 264, 302, 293, 259, 229, 203, 229,
  242, 233, 267, 269, 270, 315, 364, 347, 312, 274, 237, 278,
  284, 277, 317, 313, 318, 374, 413, 405, 355, 306, 271, 306,
  315, 301, 356, 348, 355, 422, 465, 467, 404, 347, 305, 336,
  340, 318, 362, 348, 363, 435, 491, 505, 404, 359, 310, 337,
  360, 342, 406, 396, 420, 472, 548, 559, 463, 407, 362, 405,
  417, 391, 419, 461, 472, 535, 622, 606, 508, 461, 390, 432
)
# Create date sequence
dates <- seq(as.Date("1949-01-01"), by = "month", length.out =
   length(data))
df <- data.frame(Date = dates, Passengers = data)

# --- Normalize data (MinMaxScaler equivalent) ---
min_val <- min(df$Passengers)
max_val <- max(df$Passengers)
df$Passengers_scaled <- (df$Passengers - min_val) / (max_val -
   min_val)
```

```r

# --- Function to create sequences (lags) for LSTM ---
#  transforms a time series into input-output
create_sequences <- function(series, window_size) {
  X <- list()
  y <- list()

  for (i in (window_size + 1):length(series)) {
    X[[i - window_size]] <- series[(i - window_size):(i - 1)]
    y[[i - window_size]] <- series[i]#The current value (what we want to predict)
  }

  # Convert to matrix
  X <- do.call(rbind, X) # Combine all list items into one matrix
  y <- unlist(y) #Convert the list to a vector

  return(list(X = X, y = y))#Return the result as a list containing X and y
}

# Create sequences
WINDOW_SIZE <- 12
series <- df$Passengers_scaled
sequences <- create_sequences(series, WINDOW_SIZE)
X <- sequences$X
y <- sequences$y

# --- Split into train/test (80% train) ---
# The index that separates training and testing data
split_index <- floor(nrow(X) * 0.8) #Round down
X_train <- X[1:split_index, ]
y_train <- y[1:split_index]
X_test <- X[(split_index + 1):nrow(X), ]
y_test <- y[(split_index + 1):length(y)]
length(X_test)
# --- Reshape for LSTM: [samples, time_steps, features] ---
# Convert to 3D array format expected by Keras
X_train <- array(X_train, dim = c(nrow(X_train), WINDOW_SIZE, 1))
X_test <- array(X_test, dim = c(nrow(X_test), WINDOW_SIZE, 1))
# --- Build LSTM model ---
```

```r
model <- keras_model_sequential()

model %>%
  layer_lstm(units = 64,
             activation = 'relu',
             input_shape = c(WINDOW_SIZE, 1)) %>%
  layer_dense(units = 1)

# Compile model
model %>% compile(
  optimizer = 'adam',
  loss = 'mse'
)

# Print model summary
summary(model)

# --- Train ---
history <- model %>% fit(
  x = X_train,
  y = y_train,
  epochs = 100,
  batch_size = 16,
  validation_data = list(X_test, y_test),
  verbose = 1
)

# --- Predict ---
y_pred <- model %>% predict(X_test)

# --- Inverse scale predictions back to original values ---
# Function to inverse transform
inverse_scale <- function(scaled_values, min_val, max_val) {
  return(scaled_values * (max_val - min_val) + min_val)
}

y_test_inv <- inverse_scale(y_test, min_val, max_val)
y_pred_inv <- inverse_scale(y_pred, min_val, max_val)

# --- Evaluation (RMSE) ---
rmse <- sqrt(mean((y_test_inv - y_pred_inv)^2))
```

```
cat(sprintf('RMSE: %.2f\n', rmse))

# --- Plot results ---
# Create indices for test period
test_indices <- (split_index + 1):(split_index  + length(y_test))
   +WINDOW_SIZE
test_dates <- df$Date[test_indices]

# Prepare data for plotting
plot_df <- df %>%
  select(Date, Passengers) %>%
  mutate(Type = "Original")

forecast_df <- data.frame(
  Date = test_dates,
  Passengers = as.numeric(y_pred_inv),
  Type = "LSTM Forecast"
)

combined_df <- bind_rows(plot_df, forecast_df)

# Create plot
ggplot(combined_df, aes(x = Date, y = Passengers, color = Type))
   +
  geom_line(size = 0.8) +
  scale_color_manual(values = c("Original" = "black", "LSTM
     Forecast" = "red")) +
  labs(title = "LSTM - AirPassengers Forecast",
       x = "Date",
       y = "Number of Passengers") +
  theme_minimal() +
  theme(legend.position = "bottom")
```

For co2 Dataset

```
    # Load required libraries
library(keras)
library(tensorflow)
library(ggplot2)
library(dplyr)
library(tidyr)

```

```r
# --- Load Data ---
data <- co2

# Create date sequence
dates <- seq(as.Date("1959-01-01"), by = "month", length.out =
   length(data))
df <- data.frame(Date = dates, co2 = data)

# --- Normalize data (MinMaxScaler equivalent) ---
min_val <- min(df$co2)
max_val <- max(df$co2)
df$co2_scaled <- (df$co2 - min_val) / (max_val - min_val)

# --- Function to create sequences (lags) for LSTM ---
create_sequences <- function(series, window_size) {
  X <- list()
  y <- list()

  for (i in (window_size + 1):length(series)) {
    X[[i - window_size]] <- series[(i - window_size):(i - 1)]
    y[[i - window_size]] <- series[i]
  }

  # Convert to matrix
  X <- do.call(rbind, X)
  y <- unlist(y)

  return(list(X = X, y = y))
}

# Create sequences
WINDOW_SIZE <- 12
series <- df$co2_scaled
sequences <- create_sequences(series, WINDOW_SIZE)
X <- sequences$X
y <- sequences$y

# --- Split into train/test (80% train) ---
split_index <- floor(nrow(X) * 0.8)
X_train <- X[1:split_index, ]
y_train <- y[1:split_index]
```

```
X_test <- X[(split_index + 1):nrow(X), ]
y_test <- y[(split_index + 1):length(y)]

# --- Reshape for LSTM: [samples, time_steps, features] ---
# Convert to 3D array format expected by Keras
X_train <- array(X_train, dim = c(nrow(X_train), WINDOW_SIZE, 1))
X_test <- array(X_test, dim = c(nrow(X_test), WINDOW_SIZE, 1))

# --- Build LSTM model ---
model <- keras_model_sequential()

model %>%
  layer_lstm(units = 64,
             activation = 'relu',
             input_shape = c(WINDOW_SIZE, 1)) %>%
  layer_dense(units = 1)

# Compile model
model %>% compile(
  optimizer = 'adam',
  loss = 'mse'
)

# Print model summary
summary(model)

# --- Train ---
history <- model %>% fit(
  x = X_train,
  y = y_train,
  epochs = 100,
  batch_size = 16,
  validation_data = list(X_test, y_test),
  verbose = 1
)

# --- Predict ---
y_pred <- model %>% predict(X_test)

# --- Inverse scale predictions back to original values ---
# Function to inverse transform
```

```r
inverse_scale <- function(scaled_values, min_val, max_val) {
  return(scaled_values * (max_val - min_val) + min_val)
}

y_test_inv <- inverse_scale(y_test, min_val, max_val)
y_pred_inv <- inverse_scale(y_pred, min_val, max_val)

# --- Evaluation (RMSE) ---
rmse <- sqrt(mean((y_test_inv - y_pred_inv)^2))
cat(sprintf('RMSE: %.2f\n', rmse))

# --- Plot results ---
# Create indices for test period
test_indices <- (split_index + WINDOW_SIZE + 1):(split_index +
   WINDOW_SIZE + length(y_test))
test_dates <- df$Date[test_indices]

# Prepare data for plotting
plot_df <- df %>%
  select(Date, co2) %>%
  mutate(Type = "Original")

forecast_df <- data.frame(
  Date = test_dates,
  co2 = as.numeric(y_pred_inv),
  Type = "LSTM Forecast"
)

combined_df <- bind_rows(plot_df, forecast_df)

# Create plot
ggplot(combined_df, aes(x = Date, y = co2, color = Type)) +
  geom_line(size = 0.8) +
  scale_color_manual(values = c("Original" = "black", "LSTM
     Forecast" = "red")) +
  labs(title = "LSTM - Airco2 Forecast",
       x = "Date",
       y = "cocentration of co2") +
  theme_minimal() +
  theme(legend.position = "bottom")
```

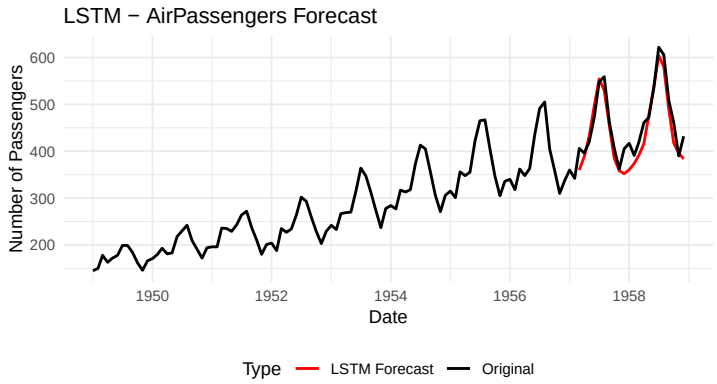


Figure 3.7: LSTM Forecasting Using AirPassengers Dataset

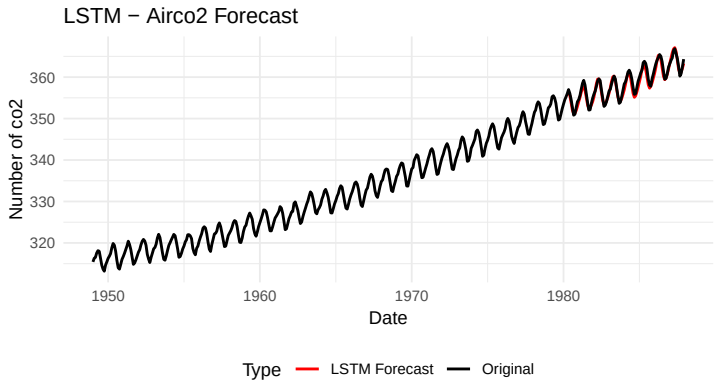


Figure 3.8: LSTM Forecasting Using co2 Dataset

From Figure 3.7 and 3.8 we remark that:

- **Overall fit**: The red forecast line closely follows the black original line for most of the time period, indicating that the LSTM successfully learned the underlying structure of the data.
- **Seasonality captured**: The model faithfully reproduces the regular yearly peaks and troughs in air travel demand (higher in summer months).
- **Trend following**: The upward trend over the years is also well captured, although slight underestimations or overestimations may appear at some extreme peaks.
- **Possible deviation at boundaries**: At the end of the series, the red line may diverge slightly from the black line, which is common in forecasting when the model extrapolates beyond the training data.

3.9 Application on Water Consumption Data of Saida-Algeria

The data used in this study were obtained during an internship conducted at the Algérienne des Eaux (ADE) unit of Saïda, which is a public institution responsible for the production and distribution of drinking water in the province of Saïda. ADE operates under the supervision of the Ministry of Water Resources and plays a major role in managing water supply services at the national level.

The Saïda unit manages water production and distribution for several municipalities within the province and has an estimated annual production capacity of approximately $13,862,890.34\, m^3$ /year. The institution relies on an organized administrative and technical structure composed of several directorates and services, including technical management, human resources management, and financial and accounting management.

The data represent quarterly water consumption observations spanning a period of 12 years, from the first quarter of 2014 to the fourthquarter of 2025, resulting in a total of 48 observations. Each observation represents the total water consumption measured in cubic meters (m^3)for a given quarter.

The collected data were used to analyze the behavior of the time series and to develop forecasting models. The objective of this study is to predict future water consumption using statistical and deep learning approaches such as ARIMA, Holt–Winters, and LSTM models.

The choice of this dataset is motivated by the importance of water demand forecasting in improving water resource management, optimizing distribution systems, and supporting decision-making within the water sector.

Year	Q1	Q2	Q3	Q4
2014	3,949,862	4,561,934	5,229,289	5,108,914
2015	4,545,364	5,261,840	5,826,494	5,090,035
2016	5,166,049	5,735,834	6,256,108	5,395,094
2017	5,211,614	5,887,956	6,269,841	5,659,095
2018	5,285,430	5,812,015	6,329,466	5,791,111
2019	5,735,042	6,260,405	6,937,581	6,473,033
2020	5,860,000	6,074,000	9,508,000	6,222,000
2021	6,076,000	6,691,000	7,662,000	6,755,000
2022	6,391,000	7,263,000	8,027,287	7,085,899
2023	6,951,044	7,694,940	8,342,368	7,677,677
2024	6,952,876	7,493,025	7,776,594	7,405,139
2025	7,268,823	7,724,089	7,965,566	7,973,685

Table 3.2: Quarterly water consumption (m^3) From ADE Saïda (2014–2025)

Data Presentation

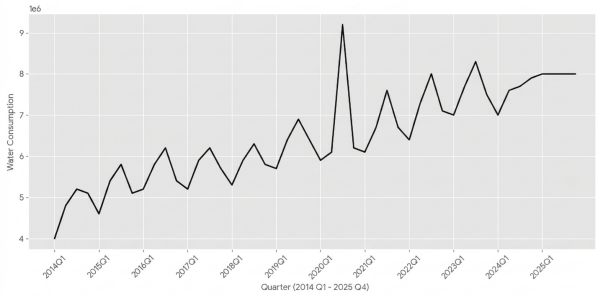


Figure 3.9: Presentation of The Data

3.9.1 R Implementation of The LSTM Model:

```
    # --- Load libraries ---
library(keras) #  for building neural networks
library(tensorflow) # which acts as the "engine" behind Keras
library(ggplot2) # for advanced graphics
library(dplyr) # for easy data manipulation
library(tidyr) # for cleaning and reshaping data
# Set seed for reproducibility
set.seed(123) # Makes results reproducible(Ensures random numbers
    are the same every time you run)
# --- Your data (48 quarterly values, 12 years) ---
data <- c(3949862, 4561934, 5229289, 5108914, 4545364,5261840,
   5826494, 5090035, 5166049, 5735834, 6256108, 5395094,5211614,
```

```
   5887956,6269841,5659095,5285430,5812015, 6329466, 5791111,
   5735042, 6260405, 6937581, 6473033,5860000, 6074000, 9508000,
   6222000,6076000, 6691000, 7662000, 6755000,6391000, 7263000,
   8027287, 7085899,6951044, 7694940, 8342368, 7677677,6952876,
   7493025, 7776594, 7405139,7268823, 7724089, 7965566, 7973685)
# --- CORRECTED: Create proper quarter labels ---
years <- rep(2014:2025, each = 4)  # 2014 to 2025 (12 years)
quarters <- rep(c("Q1", "Q2", "Q3", "Q4"), times = 12)
quarter_labels <- paste0(years, "-", quarters)
# Create dataframe
df <- data.frame(
  Quarter = quarter_labels,
  Year = years,
  Quarter_Num = quarters,
  Water = data,
  Index = 1:48
)
# --- Create a numeric time index for plotting ---
df$Time_Index <- 1:48
# --- Normalize data ---
min_val <- min(df$Water)
max_val <- max(df$Water)
df$Water_scaled <- (df$Water - min_val) / (max_val - min_val)
# --- Function to create sequences ---
create_sequences <- function(series, window_size) {
  X <- list() # Will store input sequences
  y <- list() # Will store what we want to predict(output)
  for (i in (window_size + 1):length(series)) {
    X[[i - window_size]] <- series[(i - window_size):(i - 1)]
    y[[i - window_size]] <- series[i]
  }
  X <- do.call(rbind, X)
  y <- unlist(y)
  return(list(X = X, y = y))
}
# --- Parameters (you can adjust WINDOW_SIZE) ---
WINDOW_SIZE <- 4  # Look back 1 year (4 quarters)
series <- df$Water_scaled
# Create sequences
sequences <- create_sequences(series, WINDOW_SIZE)
X <- sequences$X
```

```
y <- sequences$y
# --- Split into train/test (80/20) ---
split_index <- floor(nrow(X) * 0.8)
cat("Training samples:", split_index, "\n")
cat("Testing samples:", nrow(X) - split_index, "\n")
X_train <- X[1:split_index, ]
y_train <- y[1:split_index]
X_test <- X[(split_index + 1):nrow(X), ]
y_test <- y[(split_index + 1):length(y)]
# --- Reshape for LSTM: (samples, window_size, features) ---
X_train <- array(X_train, dim = c(nrow(X_train), WINDOW_SIZE, 1))
X_test <- array(X_test, dim = c(nrow(X_test), WINDOW_SIZE, 1))
# --- Build LSTM model (simpler for small data) ---
model <- keras_model_sequential()
model %>%
  layer_lstm(units = # 32 memory cells
               activation = 'relu',# Activation function
               input_shape = c(WINDOW_SIZE, 1)) %>% # Expect 4 time
                   steps, 1 feature
  layer_dropout(rate = 0.2) %>%  # Add dropout to prevent
     overfitting,# Randomly turns off 20% of neurons (prevents
     overfitting)
  layer_dense(units = 1) # Output layer: 1 number (prediction)
# Compile model
model %>% compile(
  optimizer = optimizer_adam(learning_rate = 0.001),
  loss = 'mse' # Mean Squared Error
)
# Print model summary
summary(model)
# --- Train model ---
history <- model %>% fit(
  x = X_train,
  y = y_train,
  epochs = 150, # Go through data 150 times
  batch_size = 8,  # Smaller batch size for small data,Update
     weights after every 8 examples
  validation_split = 0.2,  # Use 20% of training as validation,#
     Use 20% of training to check progress
  verbose = 1
)
```

```r
# --- Predict ---
y_pred_scaled <- model %>% predict(X_test)
y_pred <- y_pred_scaled * (max_val - min_val) + min_val
y_test_actual <- y_test * (max_val - min_val) + min_val
# --- CORRECTED: Get correct indices for plotting ---
# The first prediction corresponds to (WINDOW_SIZE + split_index + 1)th original data point
start_pred_index <- WINDOW_SIZE + split_index + 1
end_pred_index <- start_pred_index + length(y_pred) - 1
# Get the corresponding quarters
test_quarters <- df$Quarter[start_pred_index:end_pred_index]
test_years <- df$Year[start_pred_index:end_pred_index]
# Create prediction dataframe
pred_df <- data.frame(
  Quarter = test_quarters,
  Year = test_years,
  Actual = y_test_actual,
  Predicted = as.numeric(y_pred),
  Index = start_pred_index:end_pred_index
)
# --- Calculate MAPE ---
# DEFINE MAPE FUNCTION
mape <- function(actual, predicted) {
  # Remove any NA values
  valid_idx <- complete.cases(actual, predicted)
  actual <- actual[valid_idx]
  predicted <- predicted[valid_idx]

  # Calculate MAPE (%)
  mape_value <- (100 / length(actual)) * sum(abs((actual - predicted) / actual))
  return(mape_value)
}

lstm_mape <- mape(y_test_inv, y_pred_inv)

# --- Print predictions vs actual -
print(pred_df)
# --- Plot training loss ---
plot(history) #creates a plot showing both training and validation loss
```

```r
# --- CORRECTED PLOT: Actual vs Predicted ---
# Prepare data for plotting
plot_df <- df %>%
  select(Index, Quarter, Water) %>%
  rename(Value = Water, Type = Quarter) %>%
  mutate(Type = "Actual")
pred_plot_df <- pred_df %>%
  select(Index, Quarter, Predicted) %>%
  rename(Value = Predicted, Type = Quarter) %>%
  mutate(Type = "Predicted")
combined_plot <- bind_rows(plot_df, pred_plot_df)
# Create the plot
ggplot() +
  geom_line(data = df, aes(x = Index, y = Water, color = "Actual"
     ), size = 0.8) +
  geom_point(data = pred_df, aes(x = Index, y = Predicted, color
     = "Predicted"), size = 2) +
  geom_line(data = pred_df, aes(x = Index, y = Predicted, color =
      "Predicted"), size = 0.8) +
  scale_color_manual(values = c("Actual" = "black", "Predicted" =
      "red")) +
  labs(title = "LSTM Forecast for Quarterly Water Consumption",
       subtitle = paste("Window Size =", WINDOW_SIZE, "quarters |
          RMSE =", round(rmse, 2)),
       x = "Time Index (Each point = 1 quarter)",
       y = "Water Consumption",
       color = "") +
  theme_minimal() +
  theme(legend.position = "bottom") +
  scale_x_continuous(breaks = seq(1, 48, by = 4),
                     labels = df$Quarter[seq(1, 48, by = 4)])
# --- Alternative: Better looking plot with quarters on x-axis
   ---
ggplot() +
  geom_line(data = df, aes(x = Quarter, y = Water, group = 1,
     color = "Actual"), size = 0.8) +
  geom_point(data = pred_df, aes(x = Quarter, y = Predicted,
     color = "Predicted"), size = 2.5) +
  geom_line(data = pred_df, aes(x = Quarter, y = Predicted, group
      = 1, color = "Predicted"), linetype = "dashed") +
```

```
  scale_color_manual(values = c("Actual" = "black", "Predicted" =
      "red")) +
  labs(title = "LSTM Forecast for Quarterly Water Consumption",
       x = "Quarter (2014 Q1 - 2025 Q4)",
       y = "Water Consumption") +
  theme_minimal() +
  theme(legend.position = "bottom",
        axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_x_discrete(breaks = df$Quarter[seq(1, 48, by = 4)])  #
     Show every 4th quarter label
```

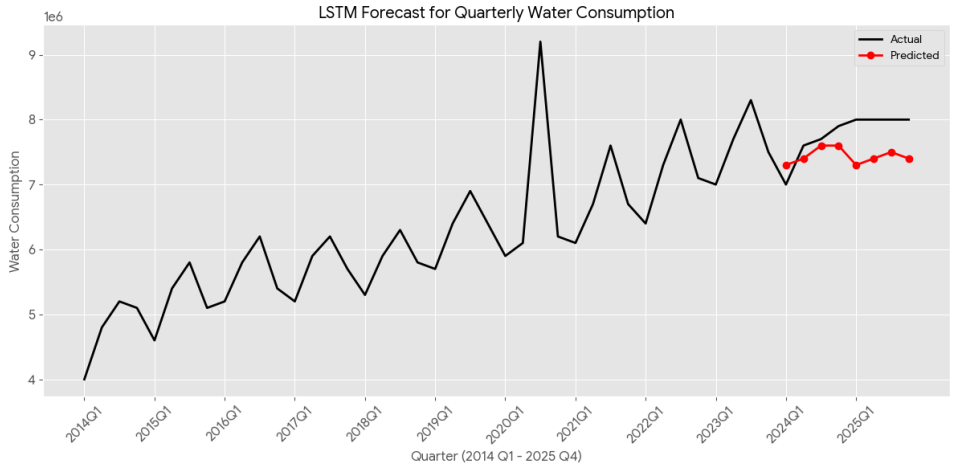


Figure 3.10: LSTM Water Consumption Forecast

3.9.2 Comparison between LSTM and Traditional Time Series Models

To provide a structured and comprehensive comparison, this section evaluates three prominent forecasting methods: Holt-Winters exponential smoothing, the Box-Jenkins (ARIMA) model, and the LSTM network. Holt-Winters is selected as a representative of classical exponential smoothing methods suited for data with trend and seasonality, while ARIMA represents the broader family of linear time series models. In contrast, LSTM serves as a deep learning alternative capable of capturing long-term dependencies and nonlinear patterns. The comparison is conducted using a real-dataset of water consumption, with actual historical values and forecasts generated by each model. The results are visualized in the accompanying graph (see figure below), which plots the actual water consumption data alongside the predictions from LSTM, ARIMA, and Holt-Winters. Using evaluation metrics such as Root Mean Squared Error (RMSE), the section assesses each model's forecasting accuracy, computational efficiency, and robustness in capturing

trends, seasonality, and irregularities in water demand patterns.

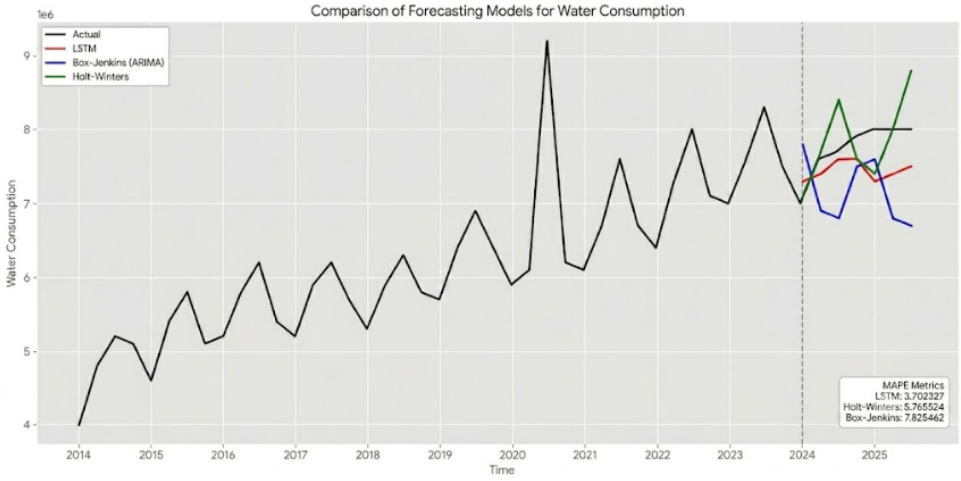


Figure 3.11: Water Consumption Forecast Comparison

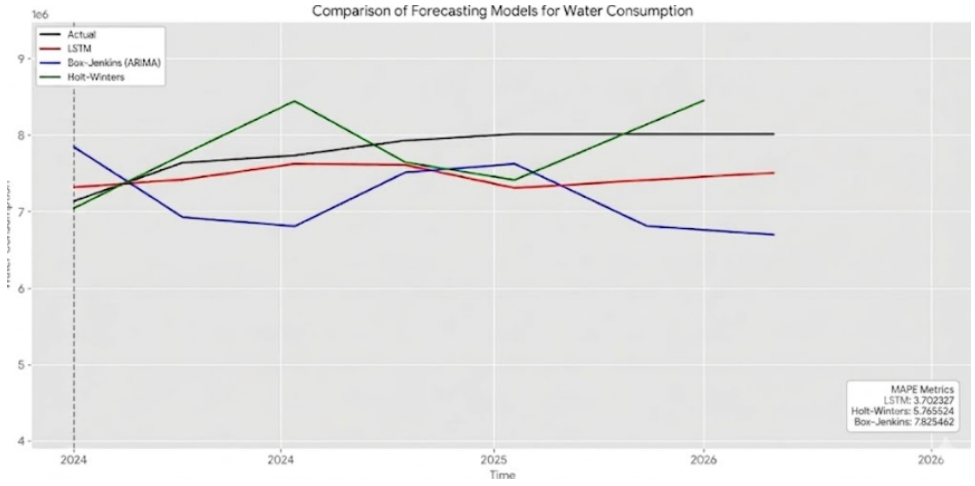


Figure 3.12: Forecast Trajectories

The results show that the LSTM model provides forecasts that are generally closer to the actual water consumption values and follows the overall trend more smoothly. This is confirmed by its lowest MAPE value (3.702327), indicating the highest prediction accuracy among the tested models.

The Holt–Winters model captures the seasonal fluctuations reasonably well and follows the increasing trend of the series, but its predictions exhibit larger deviations from the actual values compared to LSTM. Its MAPE value (5.765524) is significantly higher, reflecting lower forecasting precision.

The Box–Jenkins (ARIMA) model produces forecasts with noticeable fluctuations and underestimates several observations during the forecast horizon. It records the highest MAPE value (7.825462), suggesting the weakest performance among the three approaches.

Overall, the comparison demonstrates that the LSTM model outperforms the traditional statistical methods in forecasting water consumption, likely due to its ability to model nonlinear patterns and long-term dependencies in time series data.

3.10 Conclusion

This dissertation investigated the effectiveness of Long Short-Term Memory (LSTM) networks for time series forecasting, providing a comparative analysis against traditional statistical models. Through a combination of theoretical exposition and empirical validation on real-world data, this work has demonstrated the significant advantages of LSTM architectures for sequential data modeling, while also acknowledging their practical limitations.

Traditional time series models, including ARIMA, SARIMA, and exponential smoothing methods such as Holt-Winters, have long served as standard forecasting tools. However, these approaches struggle to capture complex nonlinear patterns and long-term dependencies that frequently characterize real-world data, particularly in non-stationary processes. Standard Recurrent Neural Networks (RNNs), while designed specifically for sequential data, suffer from fundamental limitations. The vanishing and exploding gradient problems prevent standard RNNs from learning dependencies spanning more than approximately 10–20 time steps, as empirically demonstrated by the synthetic sine wave experiment where increasing the lookback from 10 to 50 steps resulted in a nearly fourfold increase in mean squared error.

The LSTM architecture directly addresses these limitations through three key design principles. First, additive memory updates via the cell state use addition rather than multiplication, preventing gradient attenuation during backpropagation. Second, three independent gating mechanisms—the forget gate, input gate, and output gate—provide decoupled control over memory retention, writing, and reading operations. Third, the protected linear self-loop allows information to flow across arbitrarily long sequences with minimal transformation. Mathematical analysis confirms that when forget gates remain near 1 over extended periods, gradients do not vanish exponentially, in stark contrast to standard RNNs where gradient decay follows a multiplicative weight matrix path.

The cell state is not merely a "longer memory" mechanism; it fundamentally changes how sequence models can operate by decoupling the storage of information from its processing and output. This design has proven highly successful across domains including speech recognition (15), machine translation (26), and time series forecasting (21).

Empirical evidence from real-world water consumption data, obtained from the Algerienne des Eaux (ADE) Saida unit and comprising 48 quarterly observations spanning 2014 to 2025, provided concrete validation of LSTM superiority. Comparative forecasting results clearly demonstrated that the LSTM model achieved the lowest MAPE , substan-

tially outperforming Holt-Winters and Box-Jenkins ARIMA . Beyond numerical metrics, the LSTM forecast exhibited smoother trend following and closer alignment with actual consumption values, while the ARIMA model produced noticeable fluctuations and systematic underestimation of observations.

In summary, this work confirms that LSTM networks represent a substantial advancement over traditional time series forecasting methods for applications involving nonlinear patterns and long-range dependencies. The gating mechanisms and additive memory updates effectively overcome the fundamental limitations of both classical statistical models and standard RNNs. The successful implementation on real-world water consumption data demonstrates practical value for operational decision-making in public utilities, supporting improved resource planning and optimized distribution systems. Future research directions include exploring bidirectional LSTMs for enhanced context capture, attention mechanisms for focusing on relevant temporal segments, hybrid models combining statistical decomposition with deep learning, and multivariate extensions incorporating exogenous variables such as temperature, precipitation, and population growth for more comprehensive forecasting in water resource management.

Bibliography

[1] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[2] Blanchot, V. (2008, August 20). Histoire intelligence artificielle. SiecleDigital.

[3] Box, G. E. P., Jenkins, G. M., Reinsel, G. C., Ljung, G. M. (2015). *Time Series Analysis: Forecasting and Control* (5th ed.). John Wiley Sons.

[4] GeeksforGeeks. (2023, November 17). SARIMA (Seasonal Autoregressive Integrated Moving Average). Retrieved May 17, 2026, from `https://www.geeksforgeeks.org/machine-learning/sarima-seasonal-autoregressive-integrated-moving-average/`

[5] GeeksforGeeks. (2025, September 13). Difference between ANN, CNN and RNN. Retrieved from `https://www.geeksforgeeks.org/deep-learning/difference-between-ann-cnn-and-rnn/`

[6] GeeksforGeeks. (2025, August 1). Difference Between Feed-Forward Neural Networks and Recurrent Neural Networks. Retrieved from `https://www.geeksforgeeks.org/data-analysis/difference-between-feed-forward-neural-networks-and-recurrent-neural-networks/`

[7] GeeksforGeeks. (2026, April 14). Introduction to Recurrent Neural Networks. Retrieved from `https://www.geeksforgeeks.org/machine-learning/introduction-to-recurrent-neural-network/`

[8] GeeksforGeeks. (2026, March 11). Long Short-Term Memory (LSTM) using R. Retrieved April 2, 2026, from `https://www.geeksforgeeks.org/deep-learning/long-short-term-memory-lstm-using-r/`

[9] Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* (2nd ed.). O'Reilly Media.

[10] Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, volume 3, pages 189–194.

[11] Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471.

[12] Gers, F. A., Schraudolph, N. N., and Schmidhuber, J. (2001). LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340.

[13] Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning*. MIT Press.

[14] Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence, 385.

[15] Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649.

[16] Greff, K., Srivastava, R. K., Koutn'ık, J., Steunebrink, B. R., and Schmidhuber, J. (2017). LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232.

[17] Mohammed Houssou. (2005). *Optimisation de la structure des réseaux de neurones par algorithmes génétiques*. Mémoire de Magister.

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

[19] Karpathy, A., Johnson, J., and Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. In *ICLR Workshop*.

[20] Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

[21] Malhotra, P., Vig, L., Shroff, G., and Agarwal, P. (2015). Long short term memory networks for anomaly detection in time series. In *Proceedings of the 23rd European Symposium on Artificial Neural Networks*, pages 89–94.

[22] Neurone Connection. (2014). Une introduction douce à la fonction sigmoïde. Actualités de l'Intelligence Artificielle - Machine Learning - Objets connectés.

[23] Okut, H. (2021). *Deep Learning: Long Short-Term Memory*.

[24] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318.

[25] Russell, S., Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Pearson.

[26] Sutskever, I., Vinyals, O., Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 27, 3104–3112.

[27] Taleb, R. (2005). *Application des réseaux de neurones pour la prédiction de la propagation des rotules plastiques*. Mémoire de Magister.

[28] Touzet, C. (2016). *Les réseaux de neurones artificiels : Introduction au connexionnisme*.