Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Athens University of Economics and Business

Master Thesis

A Comparative Study of Traditional and Machine Learning Methods for Forecasting Time Series Data

Author:
Christina Kataki

Supervisor:
Xanthi Pedeli

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department Of Statistics

September 21, 2025

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Declaration of Authorship

I, Christina Kataki, declare that this thesis titled, "A Comparative Study of Traditional and Machine Learning Methods for Forecasting Time Series Data" and the work presented in it are my own. I confirm that:

- This work was carried out wholly or primarily during my candidature for a Master's degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed:

Date:

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

"*If you torture data long enough, it will confess to anything.*"

Ronald H. Coase

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Abstract

Department Of Statistics

Master of Science

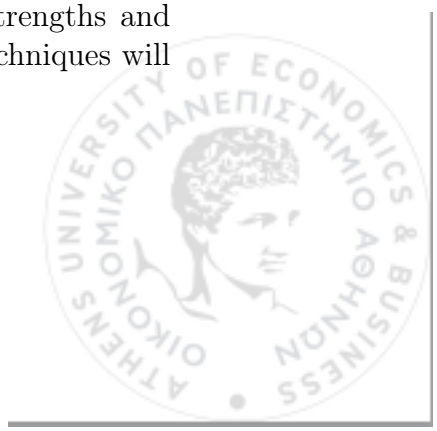
A Comparative Study of Traditional and Machine Learning Methods for Forecasting Time Series Data

by Christina KATAKI

The ability to predict future trends based on past data is crucial across various fields, from economics and finance to operations and marketing. For decades, traditional statistical models such as exponential smoothing (ETS) and autoregressive integrated moving average (ARIMA) have been the go-to tools for time series forecasting. These methods have earned their place due to their reliability, simplicity and ability to produce accurate results for many types of data. However, the rapid growth of large-scale datasets has started the adoption of advanced Machine Learning (ML) techniques which are now competing traditional models. Methods like recurrent neural networks (RNNs) introduce non-linear modeling capabilities, enabling them to capture complex patterns and subtle details that might be missed by traditional statistical models.

Despite the increasing attention and promise of ML methods, there remains a significant gap in direct comparisons between these techniques and traditional statistical approaches particularly in terms of forecasting accuracy and computational demands. While ML has great potential, it is not always evident whether it can consistently outperform traditional methods especially in practical scenarios where both accuracy and efficiency are essential.

By comparing the forecasting performance of traditional statistical models with various ML techniques, the goal is to find when and why one approach might be more effective than the other. The focus is on evaluating not just the accuracy of the forecasts but also the computational cost of these methods. Ultimately, a comparative analysis of the strengths and limitations of both traditional statistical methods and ML techniques will be provided.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Abstract

Department Of Statistics

Master of Science

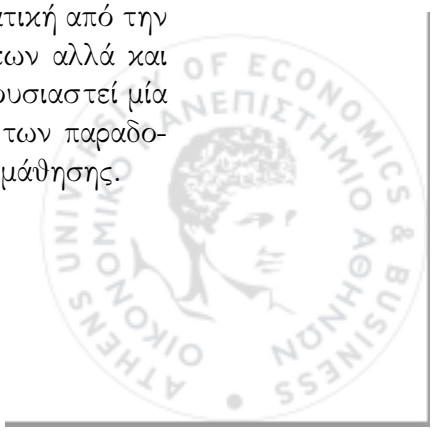
A Comparative Study of Traditional and Machine Learning Methods for Forecasting Time Series Data

by Christina KATAKI

Η δυνατότητα πρόβλεψης μελλοντικών τάσεων με βάση τα ιστορικά δεδομένα αποτελεί κρίσιμο παράγοντα σε πολλούς τομείς όπως η οικονομία, τα χρηματοοικονομικά, η επιχειρησιακή διαχείριση και το μάρκετινγκ. Για δεκαετίες, τα παραδοσιακά στατιστικά μοντέλα, όπως το μοντέλο εκθετικής εξομάλυνσης ETS (σφάλμα–τάση–εποχικότητα) και το αυτοπαλινδρομικό μοντέλο κινητού μέσου όρου (ARIMA), υπήρξαν τα κυρίαρχα εργαλεία στην πρόβλεψη χρονοσειρών. Η ευρεία χρήση τους οφείλεται στην αξιοπιστία, την απλότητα και την ικανότητά τους να παρέχουν ακριβείς εκτιμήσεις για πληθώρα διαφορετικών τύπων δεδομένων. Ωστόσο, η ραγδαία αύξηση της διαθεσιμότητας και του όγκου των δεδομένων έχει οδηγήσει στην υιοθέτηση προηγμένων τεχνικών μηχανικής μάθησης, οι οποίες πλέον ανταγωνίζονται άμεσα τις παραδοσιακές μεθόδους. Τεχνικές όπως τα επαναλαμβανόμενα νευρωνικά δίκτυα (RNNs) προσφέρουν μη γραμμική μοντελοποίηση, επιτρέποντάς τους να συλλάβουν πολύπλοκα πρότυπα και λεπτομέρειες που μπορεί να διαφύγουν από τα παραδοσιακά στατιστικά μοντέλα.

Παρά την αυξανόμενη προσοχή και τις υποσχέσεις που παρουσιάζουν οι μέθοδοι μηχανικής μάθησης, εξακολουθεί να υπάρχει σημαντικό κενό σε άμεσες συγκρίσεις μεταξύ αυτών των τεχνικών και των παραδοσιακών στατιστικών προσεγγίσεων, ιδίως όσον αφορά την ακρίβεια πρόβλεψης και τις υπολογιστικές απαιτήσεις. Αν και η μηχανική μάθηση έχει μεγάλες δυνατότητες, δεν είναι πάντα προφανές εάν μπορεί να υπερτερεί σταθερά των παραδοσιακών μεθόδων, ειδικά σε πρακτικά σενάρια όπου τόσο η ακρίβεια όσο και η αποδοτικότητα είναι απαραίτητες.

Με τη σύγκριση της απόδοσης πρόβλεψης παραδοσιακών στατιστικών μοντέλων με διάφορες τεχνικές μηχανικής μάθησης, ο στόχος είναι να διαπιστωθεί πότε και γιατί μία προσέγγιση μπορεί να είναι πιο αποτελεσματική από την άλλη. Η έμφαση δίνεται όχι μόνο στην ακρίβεια των προβλέψεων αλλά και στο υπολογιστικό κόστος αυτών των μεθόδων. Τελικά, θα παρουσιαστεί μία συγκριτική ανάλυση των δυνατών και αδύνατων σημείων τόσο των παραδοσιακών στατιστικών μεθόδων όσο και των τεχνικών μηχανικής μάθησης.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Acknowledgements

Looking back on this Master's program, I feel grateful for the journey it has been. A mix of challenges, learning and growth that has shaped me both as a student and as a person. I want to thank all my professors for sharing their knowledge so generously and for creating an environment where curiosity was encouraged and ideas were valued. Each course, each discussion, and each piece of advice has left its mark on me and will stay with me far beyond the walls of the classroom.

I am especially thankful to my supervisor, Dr Xanthi Pedeli, for guiding me through this thesis with patience, insight and encouragement. Your constructive feedback pushed me to think deeper, work harder and aim higher. I truly appreciate the time and care you dedicated to helping me bring this work to life.

And finally, to John, thank you for being the cornerstone of my life. For every late night, every moment of doubt and every small victory, you were there. Your support, understanding and ability to make me smile even in stressful moments made all the difference.

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Contents

Declaration of Authorship **ii**

Abstract **iv**

Acknowledgements **vi**

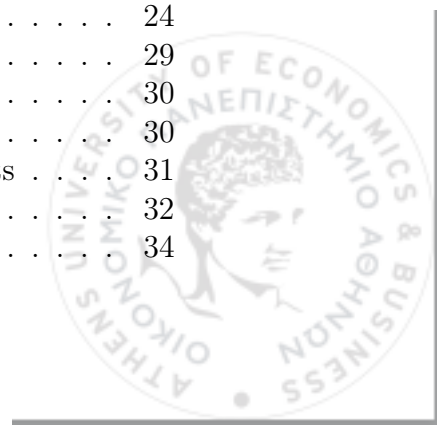
1 Introduction **1**
1.1 Thesis Structure 2

2 Time Series Essentials **4**
2.1 Time Series Analysis 4
2.2 Components of Time Series Data 5
2.2.1 Trend 5
2.2.2 Seasonality 5
2.2.3 Cyclical Variations 6
2.2.4 Irregularities 7
2.3 Mathematical Representation of Time Series 7
2.4 Statistical Analysis of Time Series 8

3 Statistical Approaches in Time Series Forecasting **11**
3.1 Seasonal Naive Forecasting 11
3.2 Exponential Smoothing Methods 12
3.3 Autoregressive Integrated Moving Average 14

4 Machine Learning Basics **16**
4.1 Decision Trees 16
4.2 Ensemble Learning 18
4.2.1 Bagging-Based Methods: Random Forests 18
4.2.2 Boosting-Based Methods: Gradient Boosting & Variants 19
4.3 Hyperparameter Tuning 22

5 Neural Networks **24**
5.1 Artificial Neural Networks 24
5.1.1 Structure of an Artificial Neuron 24
5.1.2 Multi-Layer Perceptron 29
5.2 Learning Process in Neural Networks 30
5.2.1 Initialization in Neural Networks 30
5.2.2 Training Cycle: Refining the Network Weights 31
5.2.3 Optimization Algorithms 32
5.3 Techniques to Prevent Overfitting 34



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

6 Neural Networks Strategies for Time Series Forecasting 40
6.1 Recurrent Neural Networks 40
6.1.1 Simple Recurrent Network 40
6.1.2 Long Short-Term Memory Network 46
6.1.3 Gated Recurrent Unit Network 48
6.2 Convolutional Neural Networks 49
6.2.1 Structure of Convolutional Neural Networks 50
6.2.2 One-dimensional Convolutional Neural Networks . . . 53
6.2.3 Convolution vs Cross-Correlation 55
6.2.4 Temporal Convolutional Networks 56
6.2.5 CNN-LSTM 59
6.3 Transformers 60
6.3.1 Self-Attention Mechanism 61
6.3.2 Multi-Head Attention 61
6.3.3 Positional Encoding 62

7 Practical Evaluation of the Forecasting Models 64
7.1 Evaluation Strategy 65
7.1.1 Dataset Selection 65
7.1.2 Evaluation Metric 68
7.2 Results 68

8 Conclusion and Recommendations 77

Bibliography 80

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

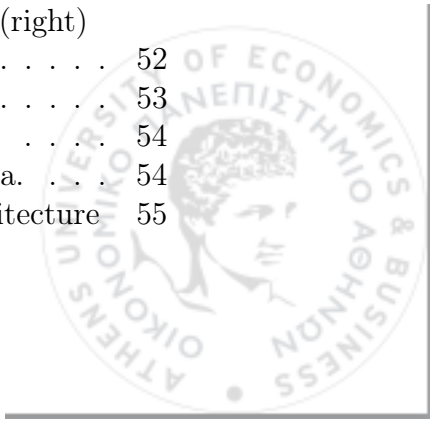
List of Figures

2.1 Global temperature trend over time from NASA GISS dataset. Source: GISTEMP Team (2024). 5
2.2 Seasonal variation in airline travel from the Air Passengers dataset. Source: Box, George E. P. and Jenkins, Gwilym M. (1976). 6
2.3 Cement production in Australia in millions of tonnes. Source: Hyndman, Rob J. and Athanasopoulos, George (2023). Business cycles in the early 1980s, 1990s, 2000s, and around 2008 are demarcated by vertical dotted lines. 7

4.1 Classification Tree: Dog vs. Cat. 17
4.2 Structure of a Random Forest 20
4.3 Comparison of Level-wise (left) vs. Leaf-wise (right) tree growth 21
4.4 Comparison of Grid search layout vs. Random search layout 22

5.1 The simplest mathematical model of a neuron called the Perceptron. 25
5.2 Linear activation functions: a. $\phi(x) = kx$, b. The identity function $\phi(x) = x$. 26
5.3 Heaviside function. 27
5.4 ReLU function. 27
5.5 Logistic function and its horizontal asymptotes. 28
5.6 Hyperbolic tangent function. 28
5.7 A Multilayer Perceptron Network 29
5.8 Dropout mechanism 36

6.1 The architecture of a Simple Recurrent Network. 41
6.2 Example of an RNN. 43
6.3 The vanishing gradient problem. 45
6.4 A standard Long Short-Term Memory architecture. 46
6.5 Long Short-Term Memory with peephole connections 48
6.6 Gated Recurrent Unit architecture. 49
6.7 Visualization of 2D convolution process. 50
6.8 Comparison of max pooling (left) and average pooling (right) operations. 52
6.9 Structure of a fully connected layer 53
6.10 Convolution operation in univariate time series data. 54
6.11 Convolution operation in multivariate time series data. . . . 54
6.12 One-dimensional Convolutional Neural Network architecture 55



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

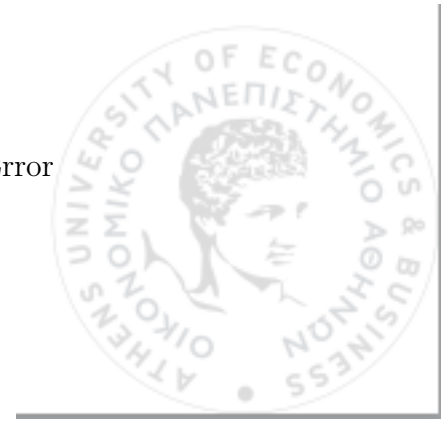
6.13 Stacked 1D convolutions. 57
6.14 Dilated convolutions. 58
6.15 The residual block in TCN architecture. 59
6.16 CNN-LSTM structure diagram. 60
6.17 Scaled Dot-Product Attention (left) and Multi-Head Attention (right). 62
6.18 Transformer-based forecasting model architecture 63

7.1 Average rank across all datasets. The darker red bars correspond to statistical methods, and the lighter peach bars correspond to neural network-based methods. 69
7.2 Rank distribution across all datasets. 70
7.3 Performance comparison of forecasting models across different temporal frequencies (Part A). 71
7.4 Performance comparison of forecasting models across different temporal frequencies (Part B). 72
7.5 Variation of TCN's ranking across different dataset selections 73
7.6 Variation of DeepAR's ranking across different dataset selections 74
7.7 Variation of Informer's ranking across different dataset selections 74
7.8 Impact of selective dataset choice on model ranking. 75

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

List of Abbreviations

A	**A**dditive
Ad	**A**dditive **d**amped
AdaGrad	**A**daptive **G**radient **A**lgorithm
Adam	**A**daptive **M**oment Estimation
ANNs	**A**rtificial **N**eural **N**etworks
AR	**A**utoregression
ARIMA	**A**utoregressive **I**ntegrated **M**oving **A**verage
BGD	**B**atch **G**radient **D**escent
BPTT	**B**ack**p**ropagation **T**hrough **T**ime
CNN	**C**onvolutional **N**eural **N**etwork
DeepAR	**Deep** **A**uto**R**egressive Model
DTs	**D**ecision **T**rees
ETS	**E**rror **T**rend **S**easonality
GRU	**G**ated **R**ecurrent **U**nit
H(x)	**H**eaviside step function
I	**I**ntegration
IQR	**I**nter**q**uartile **R**ange
KL	**K**ullback **L**eibler Divergence
LightGBM	**Light** **G**radient-**B**oosting **M**achine
LSTM	**L**ong **S**hort **T**erm **M**emory
M	**M**ultiplicative
MA	**M**oving **A**verage
MAE	**M**ean **A**bsolute **E**rror
MBGD	**M**ini-**B**atch **G**radient **D**escent
Md	**M**ultiplicative **d**amped
ML	**M**achine **L**earning
MLP	**M**ulti-**L**ayer **P**erceptron
MSE	**M**ean **S**quared **E**rror
N	**N**one
NHITS	**N**eural **H**ierarchical **I**nterpolation for **T**ime **S**eries
ReLU	**R**ectified **L**inear **U**nit
RMSE	**R**oot **M**ean **S**quared **E**rror
RMSProp	**R**oot **M**ean **S**quare **Prop**agation
RNNs	**R**ecurrent **N**eural **N**etworks
RTs	**R**egression **T**rees
SES	**S**imple **E**xponential **S**moothing
SGD	**S**tochastic **G**radient **D**escent
SMAPE	**S**ymmetric **M**ean **A**bsolute **P**ercentage **E**rror
SNaive	**S**esonal **Naive**



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

SRN	**S**imple **R**ecurrent **N**etwork
TCN	**T**emporal **C**onvolutional **N**etwork
Tanh	**H**yperbolic **T**angent
TiDE	**Ti**me-series **D**ense **E**ncoder
XGBoost	E**X**treme **G**radient **Boost**ing

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Dedicated to those who taught me and to those who kept me going.

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Chapter 1

Introduction

Forecasting has long been an important element in decision-making. Whether it involves anticipating electricity demand, projecting product sales or estimating movements in economic indicators, the ability to make informed predictions about future events directly affects efficiency and profitability. In recent decades, advances in computing power, the development of new algorithms and the wider availability of data have significantly broadened the range of tools available to forecasters. Classical statistical methods that were once the default choice now share the field with modern machine learning and deep learning approaches, each with their own advantages and limitations.

The challenge in this expanded setting is determining which forecasting approach is most appropriate for a given task. Statistical models are valued for their interpretability, computational efficiency and stability when applied to well-structured data. Machine learning methods, including neural networks, offer greater flexibility to capture complex patterns, irregular seasonality and long-range dependencies that traditional methods may find difficult to model. The question is not simply which category of method performs better overall but under what specific conditions each is most effective and whether combining different approaches can lead to further improvements.

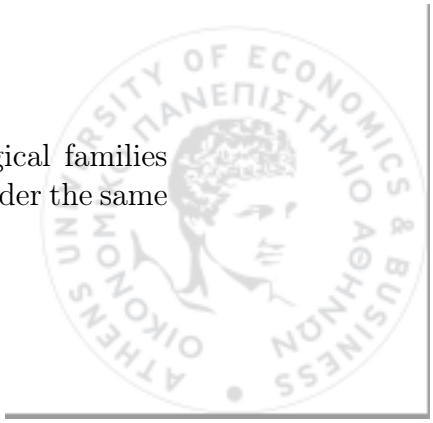
In this thesis, this question is examined in the context of univariate time series forecasting. The work combines two perspectives:

- A theoretical review of forecasting methods, spanning from traditional statistical techniques to more advanced neural network architectures, including some methods that were not tested experimentally but are relevant to understanding the field.

- An experimental comparison of selected models, conducted under consistent testing conditions to ensure that differences in performance are due to the models themselves and not to variations in the evaluation process.

The study is guided by the following questions:

1. How do forecasting methods from different methodological families perform in univariate time series tasks when evaluated under the same experimental conditions?



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

2. How do characteristics of the time series data, such as seasonality, trend and noise, influence the relative performance of different approaches?

3. How do preprocessing steps, including data transformation and normalization, affect forecasting results?

4. Can findings from the comparative experiments be used to draw practical conclusions for the selection of forecasting methods in applied scenarios?

1.1 Thesis Structure

The thesis is structured into eight chapters, each building upon the previous one:

- Chapter 2: the foundational concepts and terminology of time series analysis are presented. It describes the main components that may be present in a time series and explains how these can be represented, decomposed and analysed.

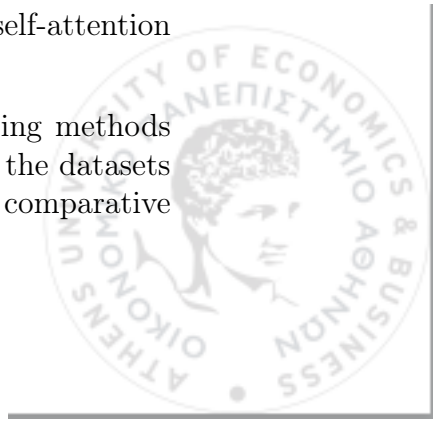
- Chapter 3: the forecasting methods grounded in statistical modelling are reviewed. It begins with simple approaches that can serve as baselines and then moves to more complex techniques, discussing the theoretical principles behind each method, their typical areas of application and the limitations that may arise in practice.

- Chapter 4: forecasting approaches derived from the broader field of machine learning are examined. It introduces the general concepts behind these methods, outlines the types of problems they are suited to address and discusses factors that influence their effectiveness, such as model complexity, training requirements and data characteristics.

- Chapter 5: the structure and operation of artificial neural networks are explained. It describes the basic computational unit (the neuron), the arrangement of neurons into layers, the role of activation functions and the mechanisms of learning through optimisation algorithms. Techniques for improving generalisation, such as regularisation and methods for controlling overfitting, are also discussed.

- Chapter 6: neural network strategies designed specifically for time series forecasting are explored. It covers recurrent neural networks, convolutional neural networks, hybrid architectures and the transformer architecture with its core mechanisms such as self-attention and positional encoding.

- Chapter 7: the experimental evaluation of the forecasting methods is presented. It explains the evaluation strategy, details the datasets used, describes the evaluation metric and reports the comparative results across the different approaches.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

- Chapter 8: the main findings of the research are summarised, their implications are discussed and the strengths and limitations of the study are highlighted. Possible directions for future work are also outlined, including extensions to the experimental setup and the exploration of alternative forecasting strategies.

The chapters follow a sequence that begins with general theoretical concepts and advances to more complex methods, moving from methodological description to experimental evaluation. This structure gives the reader the essential background and concludes with the presentation of results and their practical relevance.

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

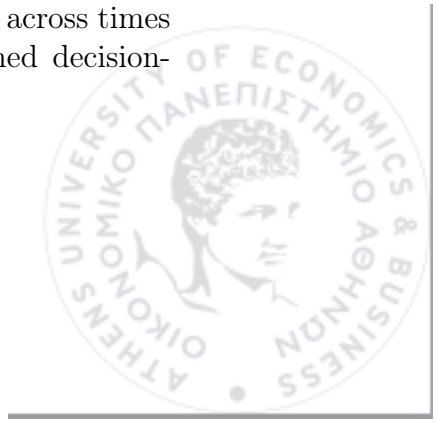
Chapter 2

Time Series Essentials

A time series is a collection of observations denoted as x_t, where each observation corresponds to a specific point in time t. It can be classified as either *discrete* or *continuous*, with discrete time series consisting of observations collected at specific, evenly spaced intervals (e.g., hourly or daily) and continuous time series involving uninterrupted observations over time. Based on predictability, time series can be further categorized as *deterministic* or *stochastic*. Deterministic time series are those that can be precisely described by mathematical rules such as $Y_t = \cos(2\pi f t)$ or $Y_t = \mu + R\sin(\omega t + \phi)$ while stochastic time series incorporate randomness where future values are characterized by probability distributions rather than exact predictions, as seen in models like $Y_t = \mu + \varepsilon_t$, where $\varepsilon_t \sim \text{i.i.d.}(0, \sigma^2)$, or $Y_t = \alpha + \beta\varepsilon_{t-1} + \varepsilon_t$ where $\varepsilon_t \sim \text{i.i.d.}(0, \sigma^2)$. Lastly, time series can be classified according to their statistical properties as *stationary* or *non-stationary*. Stationary series have constant mean and variance over time with stable periodic fluctuations, whereas non-stationary series display evolving trends or structural changes in their statistical properties. This discussion is limited to univariate time series, where each observation at time t consists of a single variable. Multivariate time series, which involve multiple interdependent variables, are not considered.

2.1 Time Series Analysis

It is essential to consider the advantages of time series analysis. At its core, it helps to understand how a variable has behaved over time by revealing important patterns and variations in the data. This information is crucial for forecasting as it allows for predicting future trends, which is a key tool for planning in business and various other sectors. Time series analysis is essential in evaluating current performance by comparing actual outcomes with expected results based on historical data. This comparison helps identify any discrepancies and their causes. Lastly, it supports comparative studies by examining changes in different phenomena across times or locations, offering valuable information that guide informed decision-making.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

2.2 Components of Time Series Data

To understand the characteristics of time series data it is essential to break it down into its components: Trend, Seasonality, Cyclical variations and Irregularities (or Noise). It is worth mentioning, that in a particular time series one or more of the aforementioned components may be absent. However, the "Irregularities" component is almost always present.

2.2.1 Trend

A trend, or long-term movement, represents the overall direction of a time series, whether it is upward or downward, shaped by factors that affect its behavior over time. It captures consistent changes, distinguishing itself from short-term variations. By focusing on this broader picture, long-term trends help identify meaningful patterns in the data. Figure 2.1 illustrates a clear upward trend in global temperature over time. This increasing trend reflects a long-term rise in global temperatures, providing a clear example of how time series analysis exposes long-term patterns that are distinct from short-term fluctuations.

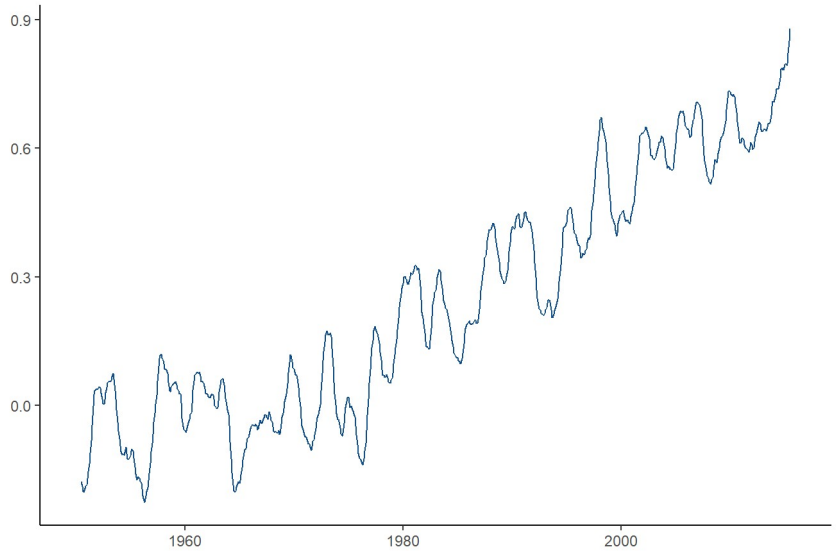
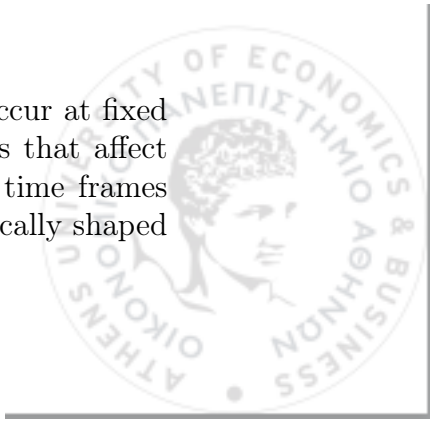


FIGURE 2.1: Global temperature trend over time from NASA GISS dataset. Source: GISTEMP Team (2024).

2.2.2 Seasonality

Seasonality refers to recurring patterns or fluctuations that occur at fixed intervals within a time series often driven by periodic factors that affect behavior over time. These patterns can manifest in various time frames such as daily, weekly, monthly or annually. Seasonality is typically shaped



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

by a wide range of external elements like weather conditions, economic cycles, cultural trends or even consumer behavior patterns. An example of seasonality can be observed in the airline industry. In a typical year, there is a seasonal variation with a clear peak appearing around the middle of the year. This pattern can be seen in Figure 2.2 where the seasonal spike in demand during the summer months is evident.

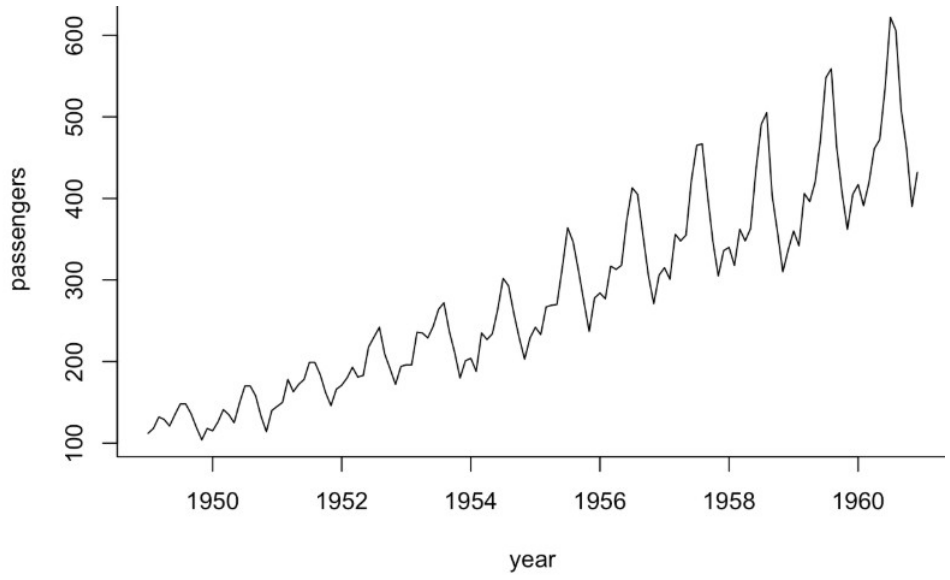
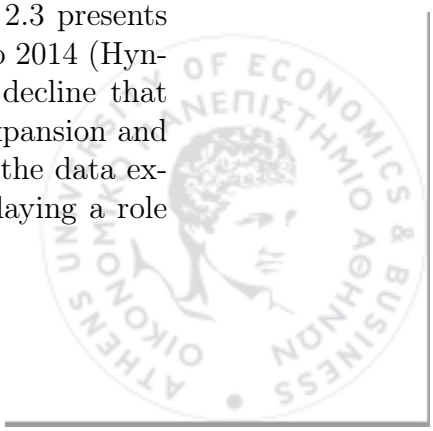


FIGURE 2.2: Seasonal variation in airline travel from the Air Passengers dataset. Source: Box, George E. P. and Jenkins, Gwilym M. (1976).

2.2.3 Cyclical Variations

Cyclical variations refer to the repetitive oscillatory movements observed in a time series over extended periods, typically lasting more than a year. These fluctuations, known as cycles, occur due to economic, environmental or structural factors and tend to exhibit variability in both duration and intensity. While the average cycle spans three to four years, some cycles can extend beyond 15 years. Unlike seasonal patterns, which follow a fixed and predictable timeline, cyclical variations do not adhere to a strict schedule and change based on broader economic or systemic conditions. Also, the magnitude of cyclical fluctuations is often more pronounced and inconsistent compared to seasonal trends. As an example, Figure 2.3 presents the time series of cement production in Australia from 1980 to 2014 (Hyndman et al., 2023), showing distinct periods of growth and decline that correspond to broader economic cycles, reflecting phases of expansion and contraction within the economy. It is worth mentioning that the data exhibits a seasonal component with weather variations likely playing a role in affecting construction activity.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

2.2.4 Irregularities

Irregular or random fluctuations, often referred to as noise, are deviations in a time series that do not follow trends, seasonal patterns or cyclical movements. These variations are erratic, unpredictable and arise from non-recurring or unforeseen events beyond human control. Factors such as unexpected external disruptions, measurement inconsistencies or random disturbances contribute to this noise. Since it lacks a structured pattern, noise makes it more difficult to identify underlying trends and accurately predict future values, introducing an inherent level of uncertainty into time series data.

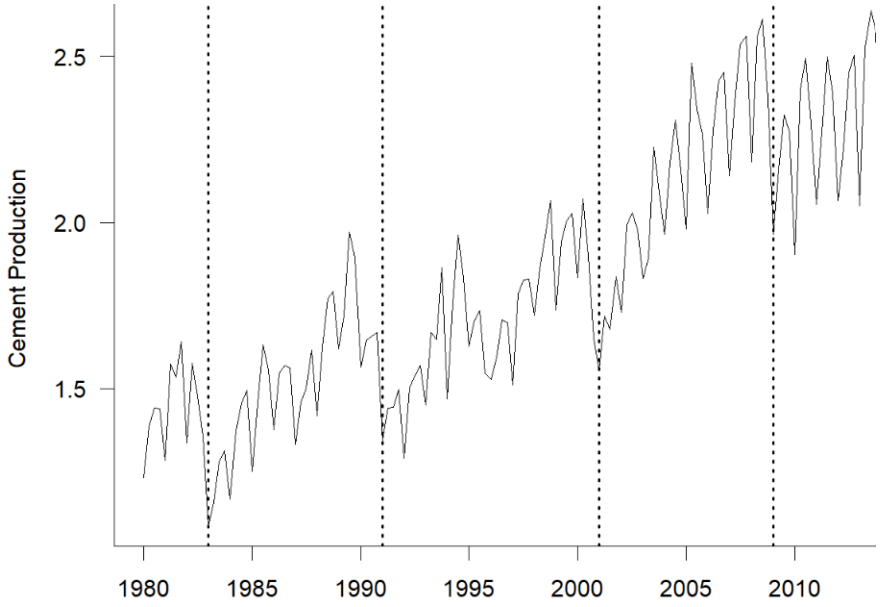
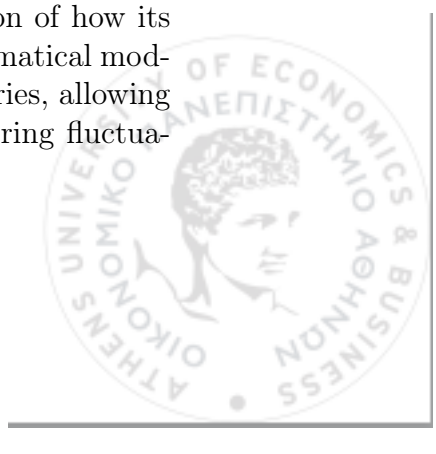


FIGURE 2.3: Cement production in Australia in millions of tonnes. Source: Hyndman, Rob J. and Athanasopoulos, George (2023). Business cycles in the early 1980s, 1990s, 2000s, and around 2008 are demarcated by vertical dotted lines.

2.3 Mathematical Representation of Time Series

Understanding time series data naturally leads to the question of how its underlying patterns can be systematically represented. Mathematical models provide a structured framework for decomposing a time series, allowing for a clearer distinction between long-term movements, recurring fluctuations and unpredictable variations.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Time series data is often modeled using one of two primary approaches:

- *Additive Model*

$$Y_t = T_t + S_t + C_t + I_t \tag{2.1}$$

- *Multiplicative Model*

$$Y_t = T_t \times S_t \times C_t \times I_t \tag{2.2}$$

where Y_t represents the original time series, while T_t, S_t, C_t, and I_t correspond to the trend, seasonal, cyclical, and irregular components, respectively.

One key advantage of the multiplicative model is that it can be converted into an additive model through logarithmic transformation:

$$\log Y_t = \log T_t + \log S_t + \log C_t + \log I_t \tag{2.3}$$

Since irregular fluctuations in a multiplicative model are strictly positive, this transformation ensures they remain mathematically well-defined while allowing for both positive and negative deviations.

The choice between these models depends on the characteristics of the time series. The additive model assumes that the components function independently, with each contributing separately to the overall behavior. However, in many practical cases the components interact dynamically, making the multiplicative model a more appropriate choice.

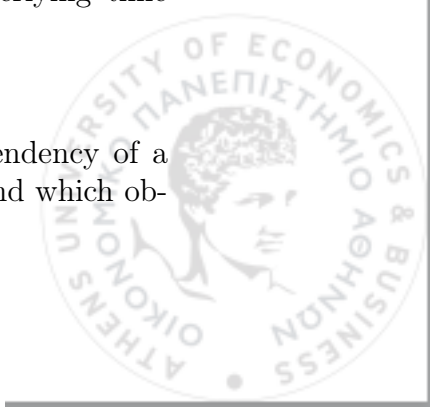
For example, in financial and economic time series, such as price indices, cyclical and seasonal variations often scale with the overall level of the data. During inflationary periods, seasonal trends become more noticeable because higher prices magnify fluctuations. On the other hand, in deflationary periods, these effects diminish as prices decline. The multiplicative model is better suited to capture such proportional relationships than the additive approach.

2.4 Statistical Analysis of Time Series

Statistical analysis provides methods for quantifying key characteristics of a time series. This facilitates the selection of appropriate and effective forecasting techniques tailored to the specific data. It is important to note that the characteristics discussed below serve as meaningful estimators of the corresponding theoretical quantities only when the underlying time series is stationary.

- **Mean Value (Expected Value)**

  The mean, or expected value, represents the central tendency of a time series and defines the long-term average level around which observations fluctuate. It is mathematically expressed as:



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

$$E[Y] = \frac{1}{N}\sum_{i=1}^{N} Y_i \tag{2.4}$$

where Y_i represents individual time series observations, and N is the total number of data points.

- **Standard Deviation**

  The standard deviation measures the dispersion of observations around the mean, representing the level of variability in a time series. It is defined as:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(Y_i - E[Y])^2}{N}} \tag{2.5}$$

  A smaller standard deviation indicates that observations are concentrated near the mean, while a larger standard deviation reflects greater dispersion and broader fluctuations.

- **Variance**

  Variance measures the extent of fluctuations in a time series and is computed as the square of the standard deviation:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(Y_i - E[Y])^2}{N} \tag{2.6}$$

- **Covariance**

  Covariance quantifies the degree to which two time series variables, X_t and Y_t, vary together over time. It is computed as:

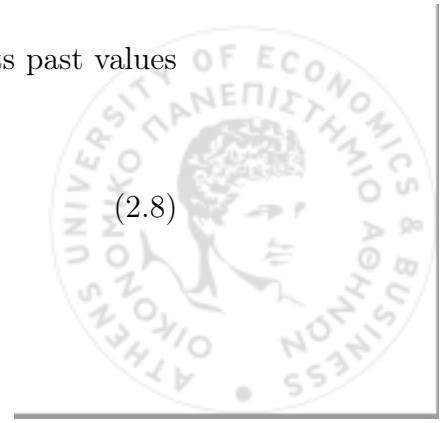
$$\gamma(X, Y) = \frac{1}{N}\sum_{t=1}^{N}(X_t - \bar{X})(Y_t - \bar{Y}) \tag{2.7}$$

  where $\bar{X}$ and $\bar{Y}$ are the mean values of X_t and Y_t, respectively. A positive covariance indicates that the two variables move in the same direction, while a negative covariance suggests they move inversely.

- **Autocovariance**

  Autocovariance measures how a time series relates to its past values at different time lags. It is given by:

$$\gamma_k = \frac{1}{N}\sum_{t=1}^{N-k}(Y_t - \bar{Y})(Y_{t+k} - \bar{Y}) \tag{2.8}$$



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

where k is the lag. A positive autocovariance indicates a direct relationship between past and future values, while a negative autocovariance suggests an inverse relationship.

- **Autocorrelation Coefficient**

  The autocorrelation coefficient is a normalized measure of autocovariance, ensuring values lie between -1 and 1. It is defined as:

$$\rho_k = \frac{\gamma_k}{\gamma_0} \tag{2.9}$$

  where γ_0 is the variance of the time series. The autocorrelation coefficient is interpreted as follows:

  – Values close to 1 indicate strong positive correlation, meaning past values are highly predictive of future values at lag k.

  – Values close to -1 suggest strong negative correlation, implying an inverse relationship.

  – A coefficient near 0 suggests little to no correlation.

  Autocorrelation is widely used in time series analysis to detect patterns, to assess dependencies and to identify seasonal components.

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Chapter 3

Statistical Approaches in Time Series Forecasting

Forecasting time series data has traditionally been accomplished using statistical models designed to capture temporal dependencies and patterns. These well-established techniques continue to be widely applied due to their efficiency and clarity. Approaches range from basic strategies such as seasonal naïve forecasting and exponential smoothing to more sophisticated models like Exponential Smoothing State Space Models and Autoregressive Integrated Moving Average. Statistical forecasting methods have shown strong performance in competitions including NN3, NN5 and M3, which benchmark methods on thousands of real-world time series from domains like finance, economics, and industry (Crone, Hibon, and Nikolopoulos, 2011; Makridakis, Spiliotis, and Assimakopoulos, 2018), where they frequently outperformed alternative approaches. However despite their effectiveness in many scenarios, these models face challenges when applied to large datasets or complex forecasting tasks. (Hewamalage, Bergmeir, and Bandara, 2021)

3.1 Seasonal Naive Forecasting

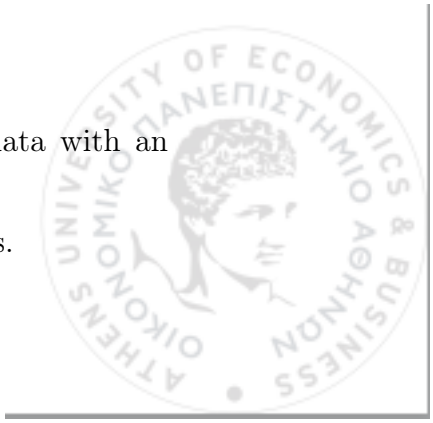
The Seasonal Naive method assumes that future values will repeat past values from the same seasonal cycle. This is effective for data with strong and stable seasonal patterns but fails when trends or random fluctuations are present.

Given a time series y_t, the forecast for a future time step $T+h$ with seasonal period m is:

$$\hat{y}_{T+h|T} = y_{T+h-km} \tag{3.1}$$

where:

- T is the last observed time point.
- h is the forecast horizon.
- m is the seasonal period (e.g., $m = 12$ for monthly data with an annual cycle).
- $k = \lfloor (h-1)/m \rfloor$ ensures alignment with seasonal cycles.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

3.2 Exponential Smoothing Methods

Exponential smoothing methods assign exponentially decreasing weights to past observations allowing them to adapt to changes over time. These methods fall under the general ETS framework, which models time series with different combinations of trend and seasonality. Simple Exponential Smoothing is a special case of ETS.

- **Simple Exponential Smoothing (SES):**
  SES is a method ideal for time series with no trend or seasonality. The SES model updates forecasts recursively (Brown, 1956):

  $$\hat{Y}_{t+1} = \alpha Y_t + (1 - \alpha)\hat{Y}_t \tag{3.2}$$

  The smoothing parameter α is interpreted as a weight between the latest observed value and the previous one-step-ahead forecast. The choice of α significantly affects how the model responds to new information:

  - If α is close to zero, the model relies heavily on past forecasts. Thus, it is more stable but slower to respond to new observations.
  - If α is close to one, the forecast is highly responsive to the latest data point, giving minimal weight to past values.
  - If α is exactly one, the SES model becomes equivalent to the naïve method (where each forecast simply repeats the most recent observation).

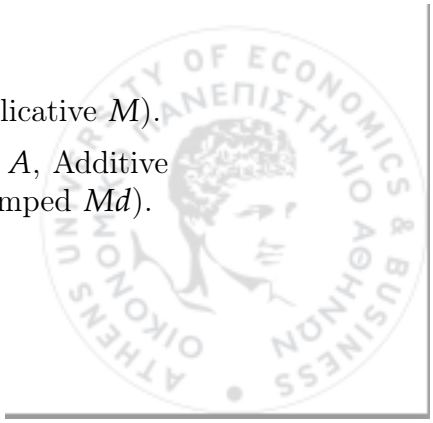
  The value of α can be selected empirically or through statistical optimization methods. The optimal choice depends on the data's characteristics and the goal of the forecasting task.

- **Exponential Smoothing State Space (ETS) Model:** The ETS model is a statistical framework used for time series forecasting. It decomposes a time series into three main components:

  - **Error (E)**: Specifies whether the error component is additive or multiplicative.
  - **Trend (T)**: Captures the long-term pattern in the data.
  - **Seasonality (S)**: Represents repeating patterns over fixed intervals.

  An ETS model is denoted as ETS(A, B, C) where:

  - A specifies the type of error (Additive A or Multiplicative M).
  - B defines the trend component (None N, Additive A, Additive damped Ad, Multiplicative M or Multiplicative damped Md).



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

– C describes the seasonal component (None N, Additive A or Multiplicative M).

Some commonly used ETS models are:

1. **ETS(A, N, N)** – Equivalent to SES
   This model corresponds to SES, as previously mentioned.
2. **ETS(A, A, N)** – Holt's Linear Trend Model
   This model extends SES by including an additive trend component.

$$y_t = \ell_{t-1} + b_{t-1} + \varepsilon_t \tag{3.3}$$
$$\ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1}) \tag{3.4}$$
$$b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1} \tag{3.5}$$

   where b_t represents the estimated trend, ℓ_t represents the smoothed level at time t and β^* determines how much weight is given to new trend updates relative to past estimates.
3. **ETS(A, Ad, N)** – Damped Trend Model
   This model includes a damping factor ϕ to gradually reduce the trend's influence over time.

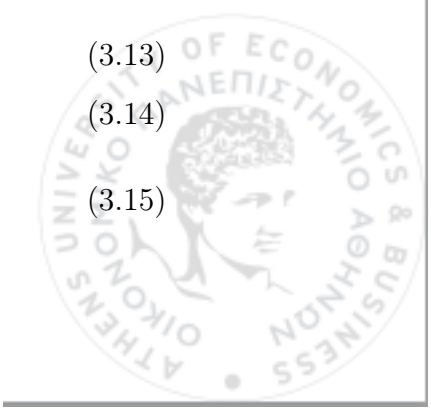
$$y_t = \ell_{t-1} + \phi b_{t-1} + \varepsilon_t \tag{3.6}$$
$$\ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + \phi b_{t-1}) \tag{3.7}$$
$$b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)\phi b_{t-1} \tag{3.8}$$

   where $0 < \phi < 1$.
4. **ETS(A, A, A)** – Holt-Winters Additive Model
   This model is used for time series with both a trend and additive seasonality.

$$y_t = \ell_{t-1} + b_{t-1} + s_{t-m} + \varepsilon_t \tag{3.9}$$
$$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1}) \tag{3.10}$$
$$b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1} \tag{3.11}$$
$$s_t = \gamma(y_t - \ell_t) + (1-\gamma)s_{t-m} \tag{3.12}$$

   where s_t represents the seasonal component of period m.
5. **ETS(A, M, M)** – Multiplicative Trend and Seasonality
   A multiplicative model is used when both the trend and seasonality vary proportionally to the level of the time series.

$$y_t = \ell_{t-1} b_{t-1} s_{t-m} \varepsilon_t \tag{3.13}$$
$$\ell_t = \alpha \frac{y_t}{s_{t-m}} + (1-\alpha)(\ell_{t-1} b_{t-1}) \tag{3.14}$$
$$b_t = \beta^* \left(\frac{\ell_t}{\ell_{t-1}} \right) + (1-\beta^*)b_{t-1} \tag{3.15}$$



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

$$s_t = \gamma \frac{y_t}{\ell_t} + (1-\gamma) s_{t-m} \tag{3.16}$$

This model is commonly used when data exhibit increasing or decreasing variance over time.

3.3 Autoregressive Integrated Moving Average

The Autoregressive Integrated Moving Average (ARIMA) model relies on the assumption that time series data possess temporal dependencies, meaning that past values affect future observations. This characteristic is particularly useful in fields such as engineering where physical processes evolve gradually rather than instantaneously. For instance, as mentioned by Box and Jenkins (1976), fluctuations in the input gas rate can affect the levels of CO_2 emitted from a furnace. In such cases, ARIMA is a suitable approach because the concentration of CO_2 does not drop to zero immediately after the gas supply is turned off. Instead, it follows a gradual decline where the rate of decrease is determined by the autoregressive coefficient ϕ_1.

The ARIMA model consists of three statistical procedures: Autoregression, Integration and Moving Average.

1. **Autoregression (AR):**

   The AR component describes the relationship between an observation and its past values. It assumes that previous observations contribute to the current value, allowing future points to be predicted based on a weighted sum of prior data. Mathematically, an AR(p) process is:

   $$X_t = c + \sum_{i=1}^{p} \phi_i X_{t-i} + \epsilon_t \tag{3.17}$$

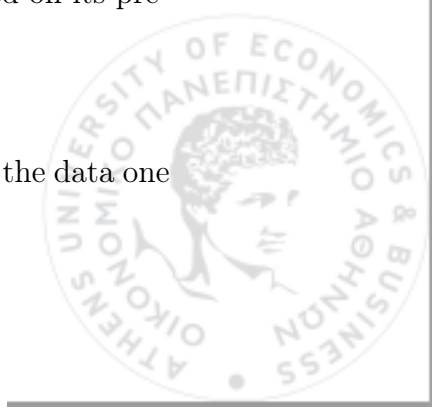
   where:

   - X_t is the value of the time series at time t,
   - c is an optional constant term,
   - ϕ_i are the autoregressive coefficients,
   - ϵ_t is a white noise error term,
   - p is the number of lagged values included in the model.

   Simply put, this component predicts a future value based on its previous values, weighted by the coefficients ϕ_i.

2. **Integration (I):**

   The I component establishes stationarity by differencing the data one or more times.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

First-order differencing is performed as follows:

$$X'_t = X_t - X_{t-1} \tag{3.18}$$

If the time series remains non-stationary after one differencing step, additional differencing can be applied:

$$X''_t = X_t - 2X_{t-1} + X_{t-2} \tag{3.19}$$

The differencing order d represents the number of times this transformation is applied:

- $d = 0$ means the data is already stationary,
- $d = 1$ means one round of differencing is applied,
- $d = 2$ means two rounds are applied, and so on.

3. **Moving Average (MA):**

   The MA component models the relationship between an observation and past forecast errors. Instead of relying directly on past values it adjusts for randomness in the data by incorporating previous prediction errors.

   Mathematically, an MA(q) process is:

$$X_t = c + \epsilon_t + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} \tag{3.20}$$

   where:

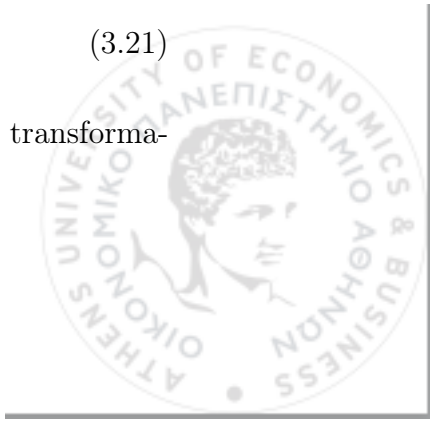
   - θ_j are the moving average coefficients,
   - ϵ_t is a white noise error term,
   - q is the number of past error terms included,
   - c is the constant term.

   This component identifies patterns in residual errors and helps correct short-term fluctuations in the model.

The complete ARIMA(p, d, q) model integrates all the aforementioned components:

$$X'_t = c + \sum_{i=1}^{p} \phi_i X'_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t \tag{3.21}$$

where X'_t represents the differenced series after applying d transformations.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Chapter 4

Machine Learning Basics

The term Machine Learning was introduced by Arthur Samuel, who described it as *a field of study that enables computers to learn without being explicitly programmed.* Unlike traditional problem-solving approaches that rely on predefined rules and algorithms, machine learning enables systems to recognize patterns and improve their performance based on data. This makes it particularly effective in solving complex problems that would otherwise be too difficult or impractical to address using conventional programming methods.

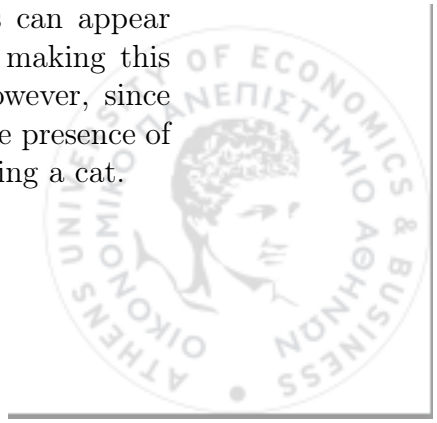
In machine learning, a model is trained on data so that it can adjust its internal parameters and improve its ability to make predictions or classifications. Thus, the model learns from the patterns within the data and adapts over time to improve its accuracy. The field of machine learning is closely linked to optimization, as many of the techniques used to train models are derived from mathematical optimization principles. These methods help improve performance and ensure that the models generalize well to new data.

Decision trees are introduced in this section as a fundamental component of tree-based ensemble models. Due to their effectiveness, they have become a prominent method in forecasting, particularly for time series data. The discussion then extends to widely used ensemble learning techniques.

4.1 Decision Trees

Decision trees (DTs) leverage input variables, commonly referred to as features in machine learning, in order to predict an outcome, known as the target variable. These predictions are generated through a structured set of decision rules that guide the categorization of data.

For example, in a classification task aimed at distinguishing between cats and dogs, the target variable corresponds to a label, either *cat* or *dog*. The distinguishing characteristics may include attributes such as nose color, ear size and other facial traits. Although pink noses can appear in both species, they are more frequently observed in cats, making this characteristic a reasonable initial classification criterion. However, since some cats also have black noses, an additional rule, such as the presence of short ears, could serve as a secondary determinant in identifying a cat.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

A decision tree is composed of three components: *root*, *branches* and *leaves*. The root node is the starting point of the tree and represents the feature that provides the most effective split in the dataset based on a chosen accuracy metric. From the root, branches extend, representing different decision points. Each branch corresponds to a rule that divides the data into groups based on whether an observation meets a specific condition. In a binary decision tree, this means each node has exactly two possible paths, sending observations in one direction if they match the condition and in another if they do not. These branching points continue to split the data into smaller and more specific groups. The final points in the tree are the leaf nodes, which contain the final classifications or predictions. Once an observation reaches a leaf, no further divisions occur.

In Figure 4.1, the root node poses the first question: *Does the animal have a pink nose?* This is the initial decision point. If the answer is yes, the classification is determined as a cat. If the answer is no, indicating that the nose is not pink, further analysis is required. The next decision node evaluates ear length: *Does the animal have long ears?* This node also branches into two outcomes. If the answer is yes, the classification is assigned as dog. If the answer is no, the classification is assigned as cat. The tree continues branching until all observations reach a leaf node, where the final classification is determined.

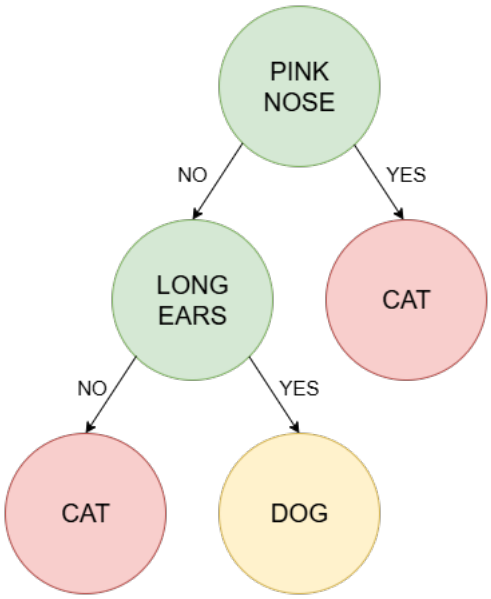
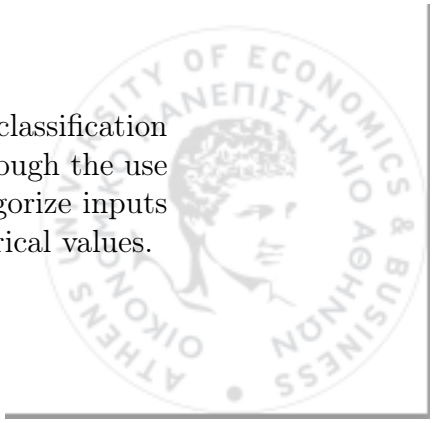


FIGURE 4.1: Classification Tree: Dog vs. Cat.

Even though DTs are widely recognized for their role in classification tasks, they can also be adapted for time series forecasting through the use of regression trees (RTs). Unlike classification trees that categorize inputs into discrete labels, RTs focus on predicting continuous numerical values.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

RTs are constructed similarly to classification trees, but the primary distinction lies in the way they split the data. Classification trees aim to separate the data into distinct categories, such as distinguishing between cats and dogs. In contrast, RTs divide the data so that the values within each group are as similar as possible. To determine the best split, RTs evaluate all available features and choose the one that creates the most homogeneous groups based on the target variable. Spiliotis (2022) uses data from the M3 forecasting competition (which contains time series belonging to 6 different domains: demographic, micro, macro, industry, finance and other) to demonstrate the application of regression trees in time series forecasting. By using features such as rolling averages, month indicators and lagged values, the study shows that decision trees can be effectively trained to capture seasonality and trends.

However, training a single RT has certain limitations, particularly its sensitivity to random fluctuations and extreme values when the number of input features is large relative to the available training data. To address this issue and improve predictive accuracy, ensemble methods have been introduced.

4.2 Ensemble Learning

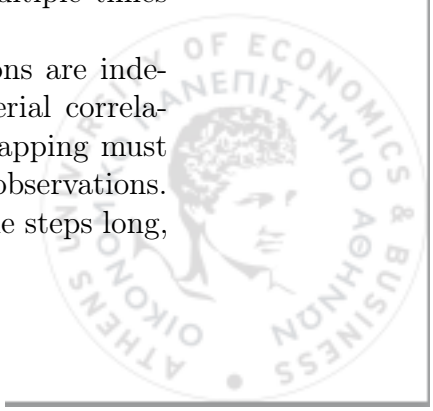
As discussed earlier, forecasting time series data is challenging, especially when relying on a single predictive model. Different models capture different aspects of the data and no single approach works best in every situation. One way to improve accuracy is by combining multiple models, as each one interprets patterns, relationships and trends in a unique way.

Ensemble learning is particularly valuable in univariate time series forecasting, where predictions are based only on past values of a single variable. Techniques such as bagging and boosting have proven useful in areas like financial forecasting and weather prediction, where small improvements in accuracy can make a significant difference.

4.2.1 Bagging-Based Methods: Random Forests

Bagging, or bootstrap aggregating, was proposed by Breiman (1996) as an ensemble method primarily designed to reduce the variance of unstable predictors. In this approach, multiple models (often DTs) are trained on different bootstrap samples drawn from the original dataset. A bootstrap sample is typically created by randomly selecting observations with replacement from the dataset, meaning that each sample has the same number of observations as the original, but some points may appear multiple times while others might be absent entirely.

However, this standard approach assumes that observations are independent, which is not the case in time series data due to serial correlation. To preserve temporal dependencies, time series bootstrapping must be adapted by resampling contiguous blocks or segments of observations. Specifically, each resampled block should be at least $d + h$ time steps long,



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

where d is the maximum lag at which serial correlation is present and h is the forecasting horizon. This ensures that both the dependency structure and the information needed for prediction are retained in each sample.

Importantly, each model is still built using the full set of features. Finally, the predictions from all models are averaged to produce the final output, as time series forecasting is inherently a regression problem. This approach reduces variance and makes the model less sensitive to outliers or biases in individual trees.

Random Forests

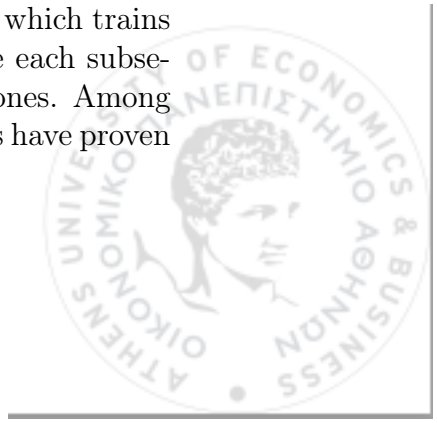
Random forests, introduced by Breiman (2001), build on the idea of bagging by adding another layer of randomness. In a random forest, each tree is still trained on a bootstrap sample, but when it comes to making decisions at each branch, the algorithm considers only a random subset of features rather than the entire feature set. This random feature selection helps to further decrease the correlation among individual trees. Figure 4.2 provides a visual representation of a random forest's structure.

It is important to note that the number of individual trees to be trained in a random forest can affect both performance and computational cost. Although random forests are easily parallelizable, since each tree can be trained independently, it's still common practice to control computational expense by limiting the depth of each tree. These shallow trees are sometimes referred to as *weak learners* due to their limited depth.

Successful applications of random forests in time series prediction can be found in various studies. Kane et al. (2014) compared ARIMA and random forest models for predicting avian influenza H5N1 outbreaks. Dudek (2015) utilized random forests for short-term load forecasting. Similarly, Lahouar and Ben Hadj Slama (2015) proposed a random forest model for one-day-ahead load forecasting. In the context of wind energy, Fischer et al. (2017) explored statistical learning approaches, including random forests, for wind power forecasting. Furthermore, Moon, Kim, and Hwang (2018) developed a hybrid short-term load forecasting method combining random forests and multilayer perceptrons.

4.2.2 Boosting-Based Methods: Gradient Boosting & Variants

Boosting is an ensemble learning technique that sequentially improves weak learners by adaptively correcting their errors. Unlike bagging, which trains models in parallel, boosting trains models sequentially, where each subsequent model focuses on correcting the errors of the previous ones. Among various boosting techniques, Gradient Boosting and its variants have proven highly effective for time series forecasting.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

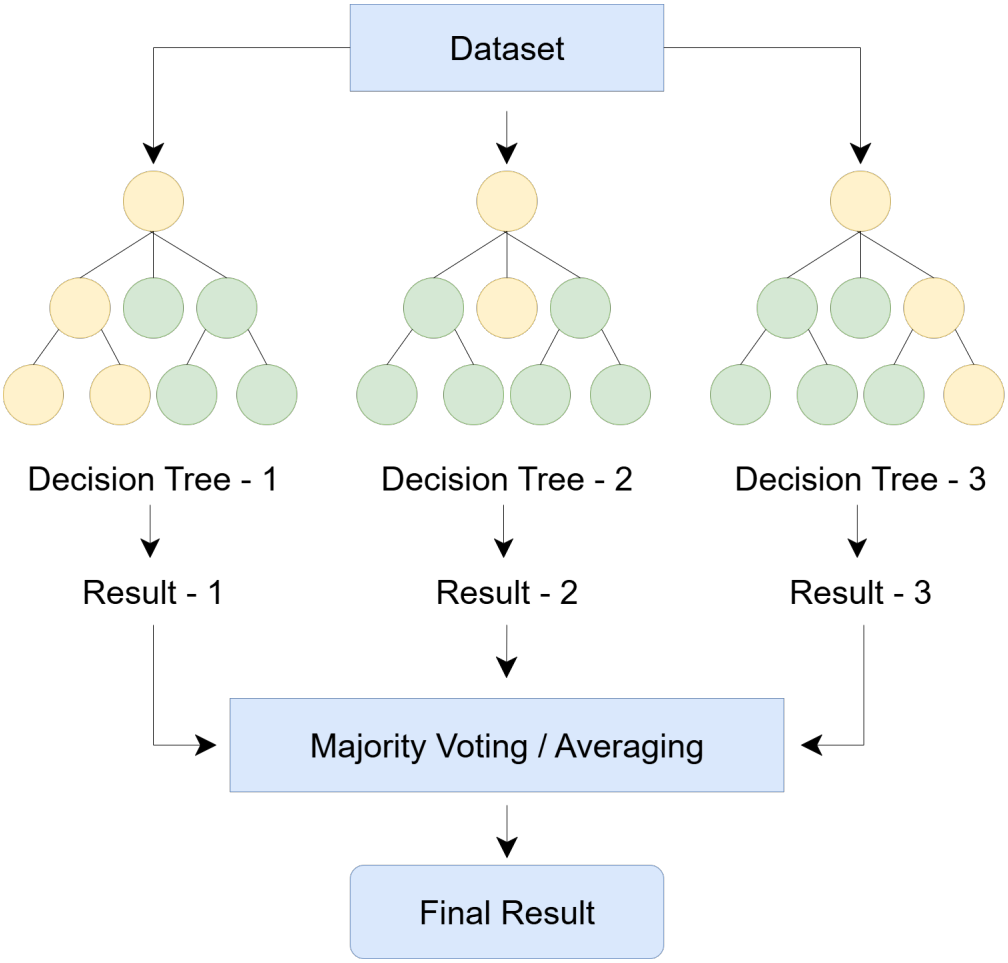


FIGURE 4.2: The structure of a Random Forest. During inference, each individual tree, having been trained on a subset of data and features, makes its own prediction. For classification tasks, the final prediction of the forest is determined by the majority vote among the trees, while for regression tasks, it is the average of all the individual tree predictions.

Gradient Boosting

Gradient boosting, originally developed by Friedman (2001), is a boosting algorithm that builds a strong predictive model by sequentially training and combining multiple weak learners, typically decision trees. At each iteration, the algorithm aims to minimize a chosen loss function (such as mean squared error in regression) by fitting a new model to the negative gradient of the loss, computed with respect to the current model's predictions. This gradient represents the direction and magnitude by which the model's current predictions should be adjusted to reduce the loss.

For example, in regression with squared error loss $\ell(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$, the negative gradient at each data point is simply the residual $y_i - \hat{y}_i$. These gradients (or residuals) form a new target variable and a DT is

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

trained to predict them. The predictions from this tree are then added (optionally scaled by a learning rate) to the model's previous predictions, resulting in updated predictions that better fit the training data. This process is repeated for a fixed number of iterations or until a stopping criterion is met, such as reaching a maximum number of iterations or when further improvements become negligible. Over time, the model becomes more accurate by sequentially correcting its own mistakes.

Within gradient boosting frameworks, decision trees are typically grown using one of two primary strategies: level-wise growth and leaf-wise growth. With *level-wise growth*, every node at a given depth is split before progressing to the next level. This method produces balanced and symmetric trees, which can help reduce overfitting (especially on smaller datasets) and tends to be easier to interpret. However, it can also be computationally demanding, as it examines all possible splits at each level, even in regions where the impact on predictions is minimal.

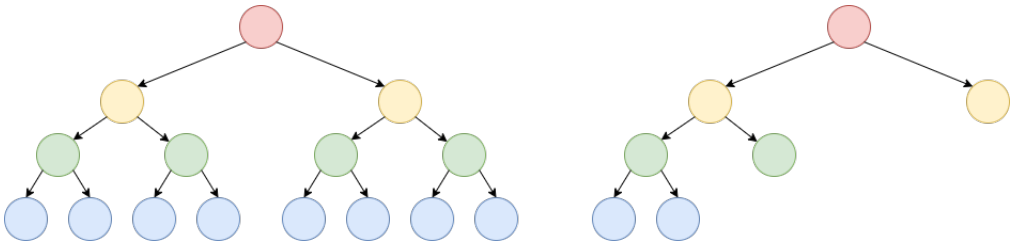


FIGURE 4.3: Comparison of Level-wise (left) vs. Leaf-wise (right) tree growth. In level-wise growth, nodes are added level by level, where a new level is only created once all nodes at the current level have been filled. This results in a balanced tree structure. In contrast, leaf-wise growth places nodes at the leaves of the tree first, leading to a less balanced, but potentially deeper, structure.

Alternatively, the leaf-wise (or best-first) strategy targets the node that currently offers the largest reduction in the loss function for splitting. This approach allows the tree to deepen in areas that require more detailed partitioning, often leading to faster convergence and improved accuracy with fewer iterations. On the downside, since this method can result in highly unbalanced trees, it raises the risk of overfitting if the tree depth or the number of trees is not adequately controlled. Figure 4.3 illustrates a comparison between level-wise and leaf-wise tree growth. Different variants of gradient boosting exist. In the context of univariate time series forecasting, two notable implementations, XGBoost and LightGBM, will be discussed.

- **Extreme Gradient Boosting (XGBoost)**

  XGBoost is one of the most popular gradient boosting frameworks, known for its speed and predictive accuracy. It includes built-in regularization to prevent overfitting and uses parallel processing to make

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

training more efficient. XGBoost primarily grows trees level-wise, which can be particularly beneficial for smaller datasets. It handles missing values by automatically determining the best direction for splits and, also, speeds up training by using a histogram-based method, that groups continuous features into discrete bins (Chen and Guestrin, 2016).

- **Light Gradient-Boosting Machine (LightGBM)**

  LightGBM is designed for speed and scalability, making it ideal for handling large datasets. It was one of the top-performing methods in the M5 forecasting competition (Makridakis, Spiliotis, and Assimakopoulos, 2022) which shows its ability to forecast multiple time series with diverse patterns. Unlike XGBoost, LightGBM grows trees leaf-wise, and this allows it to learn more efficiently and often result in faster training times and improved accuracy.

4.3 Hyperparameter Tuning

Hyperparameters are adjustable settings that shape how a model learns. They fall into two main types: model hyperparameters, which determine the structure of the learning algorithm such as tree depth or the number of estimators in ensemble models, and algorithm hyperparameters, which control training dynamics like learning rate. Unlike model parameters, which are learned from data, hyperparameters must be predefined and carefully optimized to improve predictive performance.

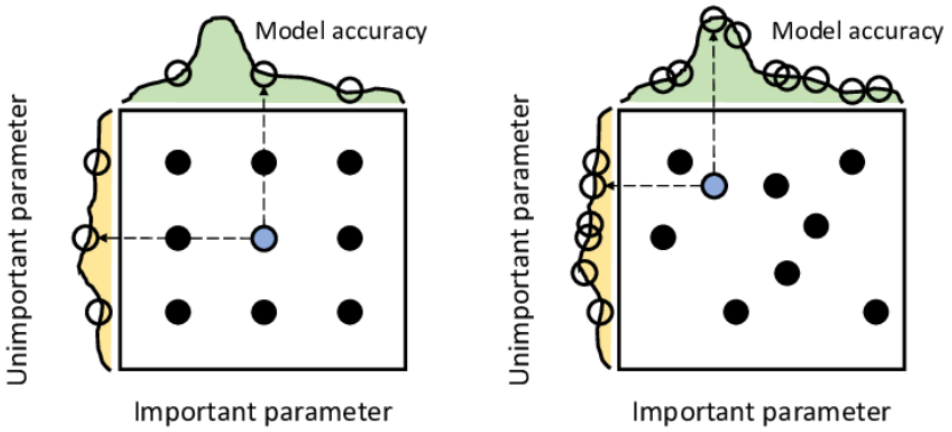


FIGURE 4.4: Comparison of Grid search layout (left) vs. Random search layout (right). Source: Pilario, Cao, and Shafiee (2020). In grid search, combinations are sampled in a structured, grid-like pattern, while in random search, values are scattered across the space. In this example, grid search misses the region containing the optimal values, which random search manages to sample.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

One of the most commonly used techniques for hyperparameter tuning is grid search, where a predefined set of hyperparameter values is systematically explored. However, this method is computationally expensive, especially for complex models like Random Forests and Gradient Boosting where multiple interacting hyperparameters must be considered.

An efficient alternative is random search, which selects hyperparameter values at random from a predefined range. This approach often finds near-optimal configurations with significantly fewer evaluations, making it more practical for tuning complex models. Figure 4.4 visually compares the structured nature of grid search with the more dispersed selection process of random search.

A more advanced technique is Bayesian optimization, which models the objective function and iteratively selects the most promising hyperparameter values. Bayesian optimization uses probabilistic models, like Gaussian processes, to focus on the most promising areas of the search space. This method significantly reduces the number of evaluations needed compared to the aforementioned methods.

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Chapter 5

Neural Networks

Neural networks are a subset of machine learning and they have become a central focus in recent research due to their exceptional performance in numerous applications. They are widely applied in many fields such as image classification, natural language processing and stock price forecasting. For example, they are important in predicting real estate prices and detecting fraudulent transactions in banking. In addition, they power recommendation algorithms which are used by e-commerce platforms to suggest products based on user preferences. In this chapter, an overview of neural networks is provided and their fundamental principles are explained.

5.1 Artificial Neural Networks

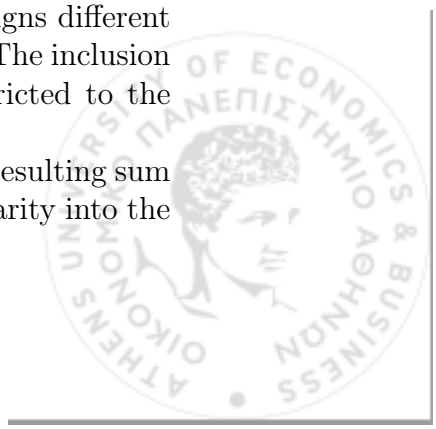
Artificial neural networks (ANNs) are computational models inspired by the way the human brain processes information. These networks consist of interconnected units, known as artificial neurons, which collectively form a system capable of learning and making predictions. The organization and connectivity of these neurons define the architecture of the network, and thus determining how data is processed.

5.1.1 Structure of an Artificial Neuron

At the core of every neural network is the artificial neuron. An artificial neuron is a fundamental processing unit that receives multiple inputs, applies a transformation and produces an output. Each input is associated with a weight which affects the neuron's decision-making process. The neuron computes a weighted sum of the inputs, adds a bias term to adjust the result and passes it through an activation function that determines the final output.

One of the simplest models of an artificial neuron is the perceptron (Figure 5.1) which is often used as a essential building block in machine learning. In this model the neuron takes an input vector, assigns different weights to each input and computes their linear combination. The inclusion of a bias term ensures that the neuron's output is not restricted to the origin.

To allow neural networks to capture complex patterns the resulting sum is passed through an activation function, introducing non-linearity into the model. The mathematical formulation of this process is:



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

$$O = f\left(\sum w_i * x_i + b\right) \tag{5.1}$$

where:

- O represents the neuron's output.
- w_i are the adjustable weights assigned to each input.
- x_i are the input values.
- b is the bias term.
- f is the activation function that transforms the weighted sum.

By adjusting the weights and biases during training, neural networks learn to recognize patterns in data.

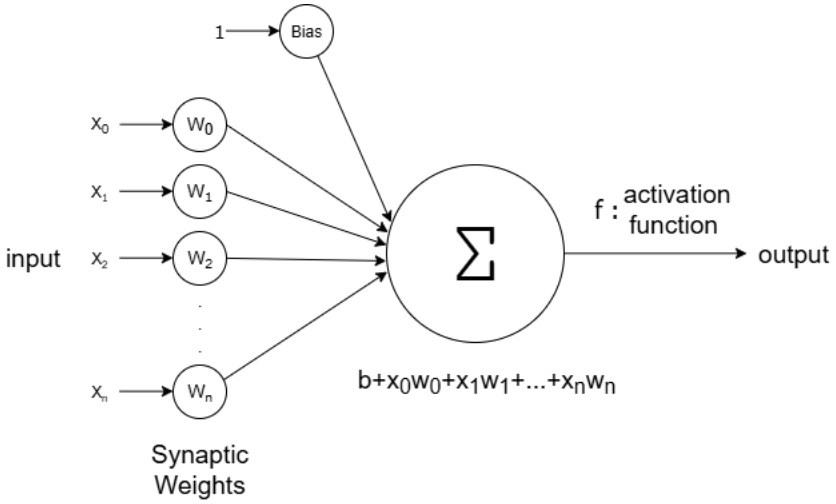
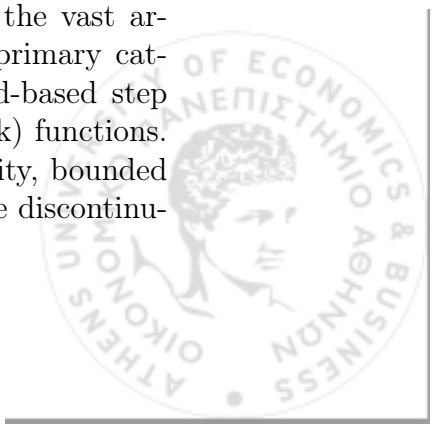


FIGURE 5.1: The simplest mathematical model of a neuron called the Perceptron.

Activation Functions

To effectively learn a nonlinear target function, neural networks use nonlinear activation functions. Recall from Equation 5.1 that the activation function $f(\cdot)$ is applied to the weighted sum of inputs and the bias. The selection of a specific activation function determines the characteristics and behavior of different types of neural networks. Among the vast array of activation functions available in the literature, three primary categories are most commonly used: linear functions, threshold-based step functions, sigmoid functions and piecewise-linear (hockey-stick) functions. Each type presents distinct advantages (such as differentiability, bounded output range or stability) but also comes with drawbacks (like discontinuities or unbounded growth).



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Examples of Activation Functions

- **Linear functions**

  A linear activation function maintains a direct proportionality between the input and output of the neuron, meaning that the transformation applied does not introduce any form of nonlinearity:

$$\phi(x) = kx, \quad k \in \mathbb{R} \tag{5.2}$$

  where k is a scaling factor (Figure 5.2 a.). This function is computationally efficient and allows direct propagation of input values. However, the major limitation is that stacking multiple layers with a linear activation function reduces the entire network to a single-layer equivalent.

  For example, the identity function $\phi(x) = x$ directly maps input values to output values without modification (Figure 5.2 b.).

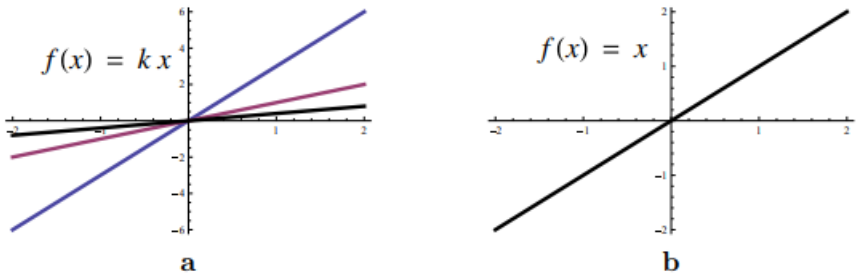


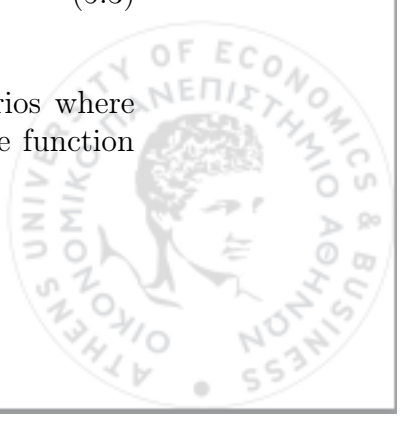
FIGURE 5.2: Linear activation functions: a. $\phi(x) = kx$, b. The identity function $\phi(x) = x$.

- **Step functions**

  Step functions, also known as threshold activation functions, activate a neuron only when the input crosses a predefined threshold. One of them is the Heaviside function (Figure 5.3) given by:

$$H(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{5.3}$$

  This function is often used in binary classification scenarios where outputs must be strictly 0 or 1. Despite its simplicity, the function suffers from non-differentiability at $x = 0$.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

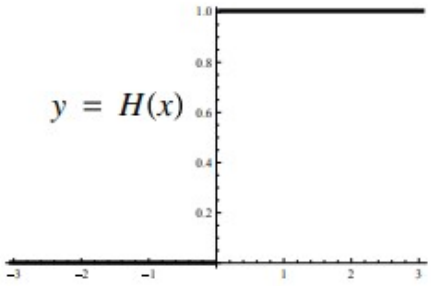


FIGURE 5.3: Heaviside function.

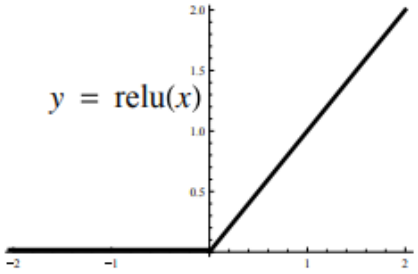


FIGURE 5.4: ReLU function.

- **Hockey-stick functions**

  These functions have graph shapes similar to an elongated right-angle curve or a hockey stick. A well-known example is the ReLU (Rectified Linear Unit) function (Figure 5.4):

$$\text{ReLU}(x) = xH(x) = \max(x, 0) \tag{5.4}$$

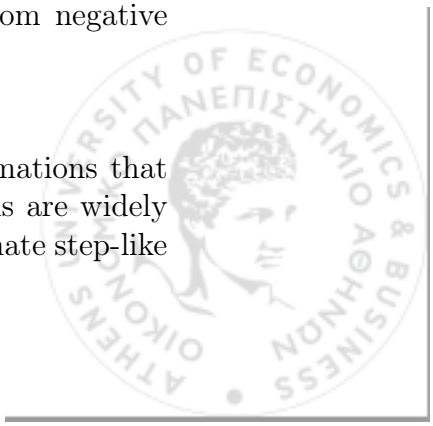
  where $H(x)$ represents the Heaviside step function. ReLU introduces sparsity by setting all negative values to zero while allowing positive values to pass through unchanged. However, one potential downside of ReLU is the dying neuron problem where negative inputs produce zero gradients which prevent weight updates. To address the dying ReLU problem, a variant known as Leaky ReLU introduces a small slope for negative inputs in order to make sure that neurons receiving negative values continue to have small and nonzero gradients:

$$\text{Leaky ReLU}(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases} \tag{5.5}$$

  where $0 < \alpha < 1$ is a small constant (typically $\alpha = 0.01$). This modification prevents entire sections of the network from becoming inactive and allows the model to retain information from negative inputs.

- **Sigmoid functions**

  Sigmoid functions introduce smooth nonlinear transformations that constrain input values to a fixed range. These functions are widely used due to their differentiability and ability to approximate step-like



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

behavior. Among the various types of sigmoid functions, the most commonly used are the following:

1. Logistic Sigmoid Function: Also known as the standard sigmoid function and is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.6}$$

This function (Figure 5.5) smoothly maps the real number domain onto the interval $(0, 1)$ making it particularly useful for probabilistic interpretations in classification problems. However, it suffers from vanishing gradients which can slow down learning.

2. Hyperbolic Tangent Function (Tanh): This is another widely used sigmoid function, mathematically expressed as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{5.7}$$

Unlike the logistic function, tanh (Figure 5.6) is zero-centered, mapping real numbers into the interval $(-1, 1)$ in order to improve training by reducing bias in weight updates.

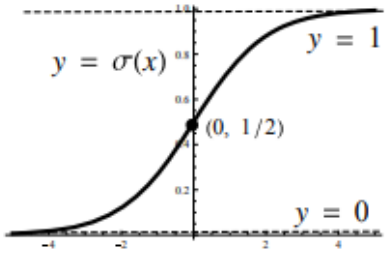


FIGURE 5.5: Logistic function and its horizontal asymptotes.

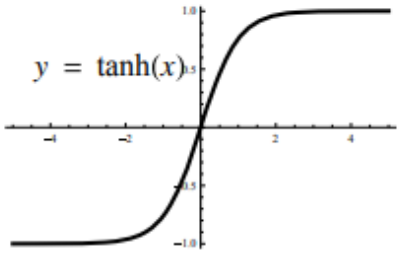
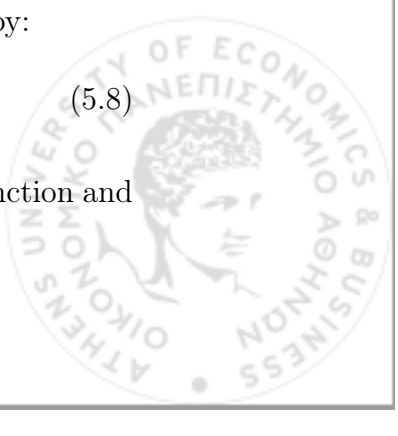


FIGURE 5.6: Hyperbolic tangent function.

3. Softsign Function: A variation of tanh with polynomial decay instead of exponential is the softsign function, given by:

$$\text{softsign}(x) = \frac{x}{1 + |x|} \tag{5.8}$$

This function retains all the advantages of the tanh function and also smooths out large input values more gradually.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

4. Piecewise-Linear Function: Another category of sigmoid-like functions consists of piecewise-linear functions, defined as:

$$f_\alpha(x) = \begin{cases} -1, & x \leq -\alpha \\ \frac{x}{\alpha}, & -\alpha < x < \alpha \\ 1, & x \geq \alpha \end{cases} \tag{5.9}$$

where α determines the slope of the central linear region. As $\alpha \to 0$, this function approaches a step function.

5.1.2 Multi-Layer Perceptron

Until the 1980s, all neural network architectures had only a few layers. The first attempt to model a biological neuron was made by McCulloch and Pitts (1943) but their model had significant learning limitations. In 1958, Rosenblatt introduced the Multi-Layer Perceptron (MLP) (Rosenblatt, 1958).

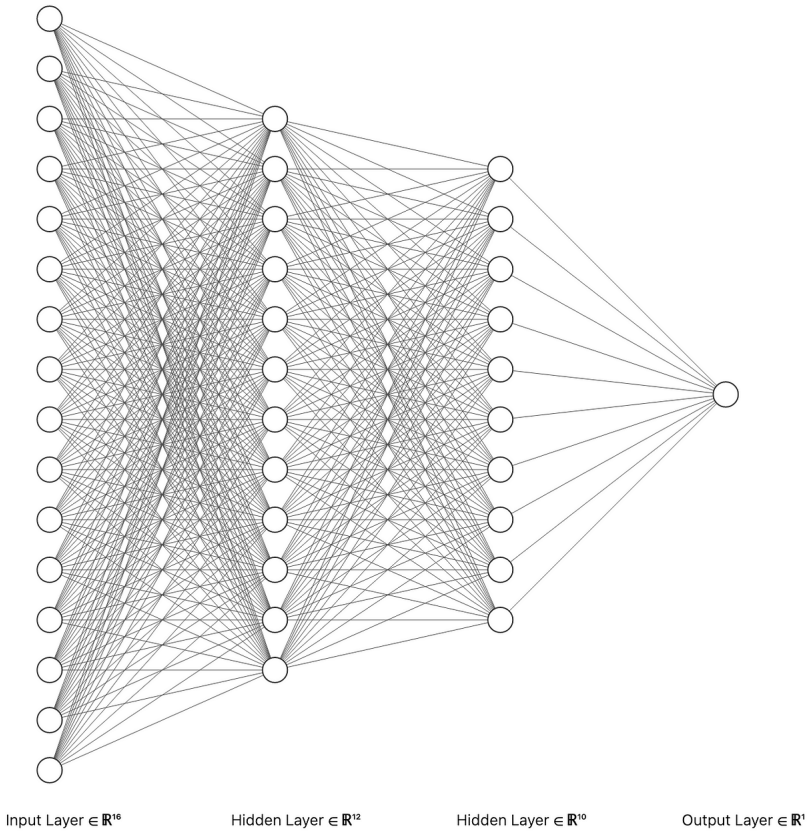
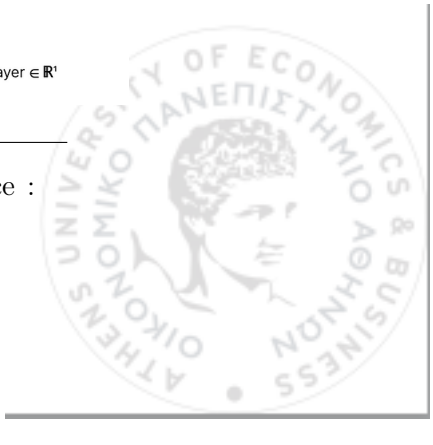


FIGURE 5.7: A Multilayer Perceptron Network. Source : Lenail (2019).



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

MLPs belong to the broader class of feedforward neural networks (FNNs), meaning that information moves strictly from the input layer towards the output layer, without any cycles or feedback loops. In other words, the output of a neuron is never fed back to itself or a previous layer. The architecture of the MLP is fully defined by an input layer, one or more hidden layers and an output layer, as illustrated in Figure 5.7. Hidden layers are the intermediate layers between the input and output layers. Each layer consists of at least one neuron. The input is processed in a forward direction, passing through each hidden layer sequentially before reaching the output.

Based on the Universal Approximation Theorem, a feedforward neural network with at least one hidden layer containing a sufficient number of neurons can approximate any continuous function on a compact domain, provided that an appropriate choice of weights and activation functions is used. This theorem demonstrates the power of MLPs and their ability to model complex and nonlinear relationships in data.

5.2 Learning Process in Neural Networks

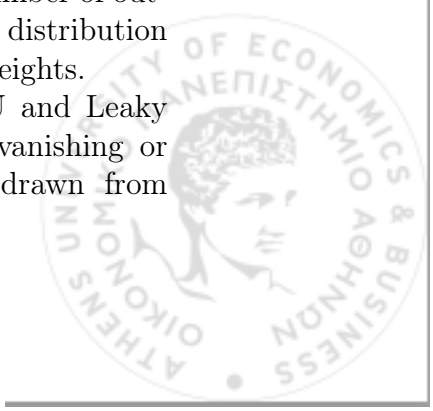
Training a neural network requires a dataset as input and a loss function that quantifies prediction errors. The network iteratively adjusts its parameters in order to minimize this loss and improve its ability to generalize to unseen data. In many cases, training data is accompanied by the correct outputs, guiding the learning process. This approach known as supervised learning enables the network to refine its predictions by continuously comparing them to the expected results and updating its internal parameters accordingly.

5.2.1 Initialization in Neural Networks

Initialization in a neural network sets the initial values of the weights and biases (such as w_i and b in Equation 5.1). Zero initialization causes symmetry issues which leads all neurons in a layer to learn identical features and prevents effective learning. In contrast, random initialization helps break this symmetry.

Xavier initialization (Glorot and Bengio, 2010) is designed for networks using sigmoid and tanh activations but can also be used for ReLU. It scales weights based on the number of input and output neurons to maintain stable activations and gradients. Formally, weights are drawn from a uniform distribution $U[-a, a]$ with $a = \sqrt{6/(n_{in} + n_{out})}$. Here, n_{in} (fan-in) is the number of inputs to a neuron/layer and n_{out} (fan-out) is the number of outputs. Since the input and output sizes vary across layers, this distribution is computed separately for each layer and its corresponding weights.

He initialization (He et al., 2015) is optimized for ReLU and Leaky ReLU activations using a higher scaling factor to prevent vanishing or exploding gradients. In this case, weights are typically drawn from $\mathcal{N}(0, 2/n_{in})$.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Biases are typically initialized to zero or small values. The choice of initialization method depends on the activation function and the architecture of the network.

5.2.2 Training Cycle: Refining the Network Weights

After a neural network is initialized, it undergoes an iterative training process to optimize its parameters. This process adjusts the weights and biases of the network to minimize the error between predicted and actual values. At the core of this learning cycle is the backpropagation algorithm, which computes the gradients of the loss with respect to the network's parameters by propagating error signals backward through the layers. The parameters are then updated using these gradients to reduce the loss.

During training, input data is passed through the network in a process known as forward propagation, where each layer applies its weights and activation functions to produce an output prediction. A loss function is then used to evaluate how far the predicted output deviates from the expected target values. Backpropagation follows, using the chain rule of differentiation to calculate how each parameter contributed to this error, so that the optimizer can update them accordingly. A more detailed description of backpropagation, specifically Backpropagation Through Time for recurrent networks, is provided in subsection 6.1.1.

Training data is typically divided into smaller subsets known as batches. Instead of updating the weights after each individual data point, the network processes a batch of samples in parallel and accumulates gradient information before performing an update, a method known as mini-batch training. The prediction error (training loss) is computed using the same batch that was just processed. To monitor generalization and detect overfitting, a separate validation set is usually evaluated between epochs or at the end of training.

Algorithm 1 Training Cycle

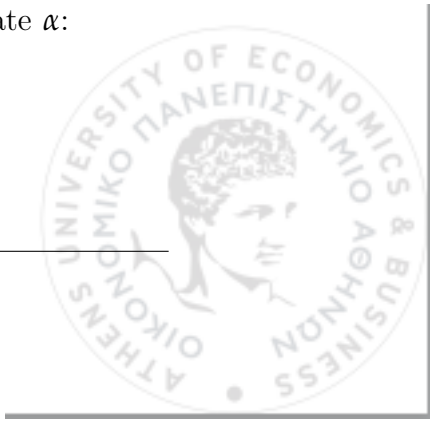
for epoch in range(num_epochs) **do**
  for batch (X, Y) in Dataset **do**
    Forward pass: Compute model predictions $\hat{Y} = f(X; \mathbf{w})$
    Compute loss: $L = \mathcal{L}(\hat{Y}, Y)$
    Backward pass: Compute gradients of loss w.r.t. each weight:

$$\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_n}$$

    Optimization step: Update weights using learning rate α:

$$w_i \leftarrow w_i - \alpha \cdot \frac{\partial L}{\partial w_i}, \quad \text{for each } i = 1, \ldots, n$$

  end for
end for



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

One complete pass through the entire training dataset is referred to as an epoch. The network iterates through multiple epochs, progressively refining its parameters until the loss function stops decreasing or improves only insignificantly, showing that the model has learned the underlying patterns in the data.

Algorithm 1 shows the typical training cycle in a neural network. Through the cycle's iterations, the model gradually improves its predictions on the target values of the training data. With a sufficiently rich and representative dataset, and provided overfitting is avoided, the model can achieve similar performance on unseen data.

5.2.3 Optimization Algorithms

The primary goal of optimization algorithms is to efficiently update the neural network's parameters (weights and biases) in a way that minimizes the loss function.

Gradient Descent Algorithm

The gradient descent algorithm is one of the most commonly used optimization methods. Given an objective function $h(\theta)$, typically defined as the average loss over a dataset, the update rule for the model's parameters is given by:

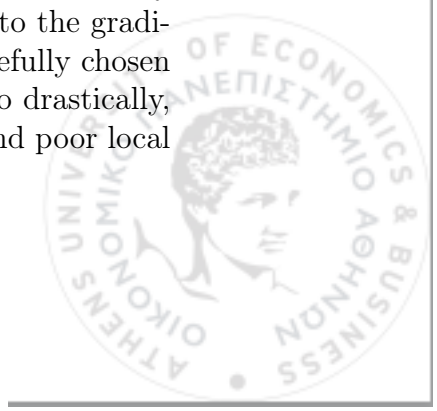
$$\theta_{t+1} = \theta_t - \alpha \nabla h(\theta_t), \tag{5.10}$$

where:

- θ_t represents the set of model parameters at time t,
- α is the learning rate, controlling the step size of updates,
- $\nabla h(\theta_t)$ represents the gradient of the objective function with respect to the parameters.

Some variants are:

1. **Batch Gradient Descent (BGD):** Uses the entire dataset to compute the gradient in each iteration, leading to more stable but slower updates. However, it can be computationally expensive for large datasets.
2. **Stochastic Gradient Descent (SGD):** Updates parameters using the gradient computed from a single training example. While it may seem counterintuitive, this approach introduces noise into the gradient estimate that can actually be beneficial. With a carefully chosen learning rate that prevents the model from changing too drastically, this noise can help the optimizer escape saddle points and poor local minima, potentially leading to faster convergence.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

3. **Mini-batch Gradient Descent:** Uses small batches of data for each update, balancing the stability of BGD with the speed of SGD.

Momentum-Based Gradient Descent

Momentum-based gradient descent improves standard gradient descent by incorporating past gradients into the update step.

$$v_{t+1} = \mu v_t - \eta \nabla f(x_t) \tag{5.11}$$

$$x_{t+1} = x_t + v_{t+1} \tag{5.12}$$

where:

- v_t is the velocity (exponentially smoothed past gradients),
- μ is the momentum coefficient (typically set to 0.9), controlling the contribution of past gradients,
- $\nabla f(x_t)$ is the current gradient of the objective function,
- η is the learning rate.

Adaptive Gradient Methods

Several modern optimizers adjust the learning rate dynamically:

- **AdaGrad**

  The Adaptive Gradient Algorithm (AdaGrad) adapts the learning rate for each parameter individually. Let $g_\tau = \nabla_\theta h(\theta_\tau)$ denote the gradient of the loss function with respect to the parameters at time step τ. The algorithm accumulates the outer products of past gradients:

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^T \tag{5.13}$$

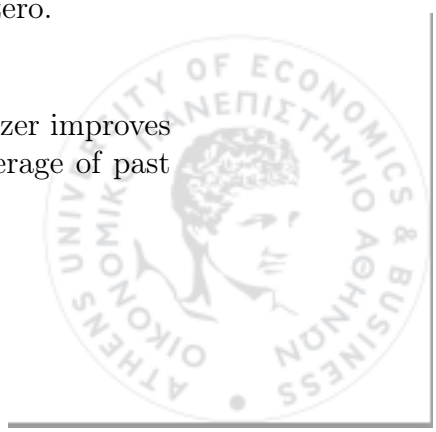
  The parameter update rule is as follows:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\text{diag}(G_t)} + \epsilon} \nabla h(\theta_t) \tag{5.14}$$

  where ϵ is a small constant added to avoid division by zero.

- **RMSProp**

  The Root Mean Square Propagation (RMSProp) optimizer improves AdaGrad by maintaining an exponentially decaying average of past squared gradients:



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla h(\theta_t))^2 \tag{5.15}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t} + \epsilon} \nabla h(\theta_t) \tag{5.16}$$

where β is typically set to 0.9.

- **Adam**

  The Adaptive Moment Estimation (Adam) optimizer combines momentum and RMSProp:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla h(\theta_t) \tag{5.17}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla h(\theta_t))^2 \tag{5.18}$$

  Bias correction:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{5.19}$$

  Update rule:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{5.20}$$

  where:

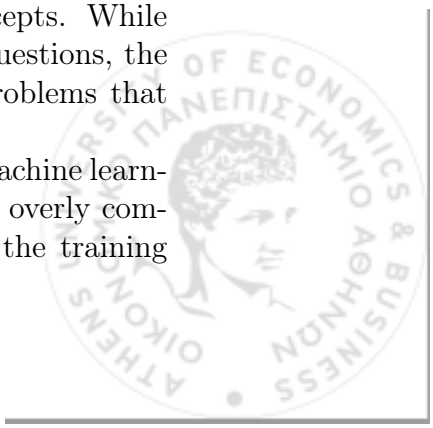
  - m_t is the first moment estimate (mean of gradients),
  - v_t is the second moment estimate (uncentered variance),
  - β_1, β_2 are decay rates (typically 0.9 and 0.999, respectively),
  - ϵ is a small constant to prevent division by zero.

5.3 Techniques to Prevent Overfitting

"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."
- John von Neumann -

Imagine a student who prepares for an exam by memorizing the answers to past questions without understanding the underlying concepts. While this approach might lead to perfect scores on those specific questions, the student is likely to struggle when faced with new, unseen problems that require deeper comprehension.

This scenario is analogous to the problem of overfitting in machine learning. More formally, overfitting occurs when a model becomes overly complex and captures not only the true underlying patterns in the training



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

data but also the random noise. As a result, the model may fit the training data extremely well but fail to generalize to new, unseen data, much like the student who memorizes rather than learns.

To overcome overfitting, various techniques have been developed to improve a model's ability to generalize to new data. Some of these techniques are outlined below.

Regularization

Regularization introduces additional information to prevent overfitting by penalizing extreme parameter weights during the model's training process. This is typically achieved by modifying the loss function to include a regularization term, which discourages the model from fitting the noise in the data.

- **L2 Regularization**

  L2 regularization, commonly referred to as Ridge Regression, involves adding a penalty term to the loss function that is proportional to the square of the magnitude of the coefficients. The objective function for a model with L2 regularization is:

  $$L_{\text{reg}} = L + \frac{\alpha}{2}\|w\|_2^2 \tag{5.21}$$

  where:

  - L is the original cost function (e.g., mean squared error),
  - α is a positive regularization parameter that controls the strength of the penalty,
  - $\|w\|_2^2$ represents the sum of the squares of the coefficients.

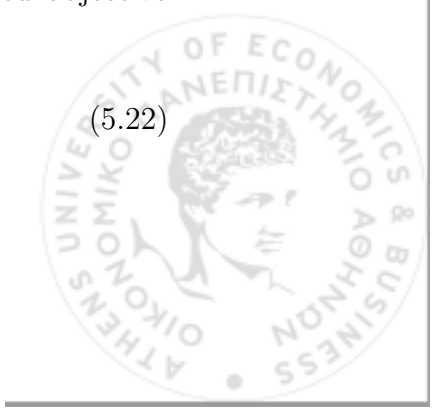
  The primary effect of L2 regularization is to modify the learning rule to shrink the weight vector multiplicatively by a constant factor during each update, effectively pulling the weights towards zero without making them exactly zero. This helps in reducing model complexity and variance. In practice, the value of α is chosen through cross-validation by evaluating model performance over a range of candidate values on a validation set.

- **L1 Regularization**

  L1 regularization, or LASSO, adds a penalty equal to the sum of the absolute values of the model's coefficients. The modified objective function is:

  $$L_{\text{reg}} = L + \alpha\|w\|_1 \tag{5.22}$$

  where:



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

- $\|w\|_1$ is the L1 norm, i.e., the sum of the absolute values of the coefficients,
- α is a regularization hyperparameter that controls the strength of the penalty.

Unlike L2 regularization, which penalizes large weights, L1 regularization tends to produce sparse solutions by driving some coefficients to exactly zero. This effectively performs feature selection and is particularly useful in high-dimensional settings. Optimal values of α are commonly determined via cross-validation, much like in L2 regularization.

- **Dropout**

  Dropout (Srivastava et al., 2014) is a regularization technique used to prevent overfitting in neural networks. During training, it works by randomly deactivating a subset of neurons in the network. This forces the model to avoid relying too heavily on specific neurons, encouraging it to learn more robust and generalized features. Dropout is applied only to the training data. During inference (i.e., evaluation on validation or test data) all neurons remain active and their outputs are scaled accordingly to maintain consistency with the training distribution.

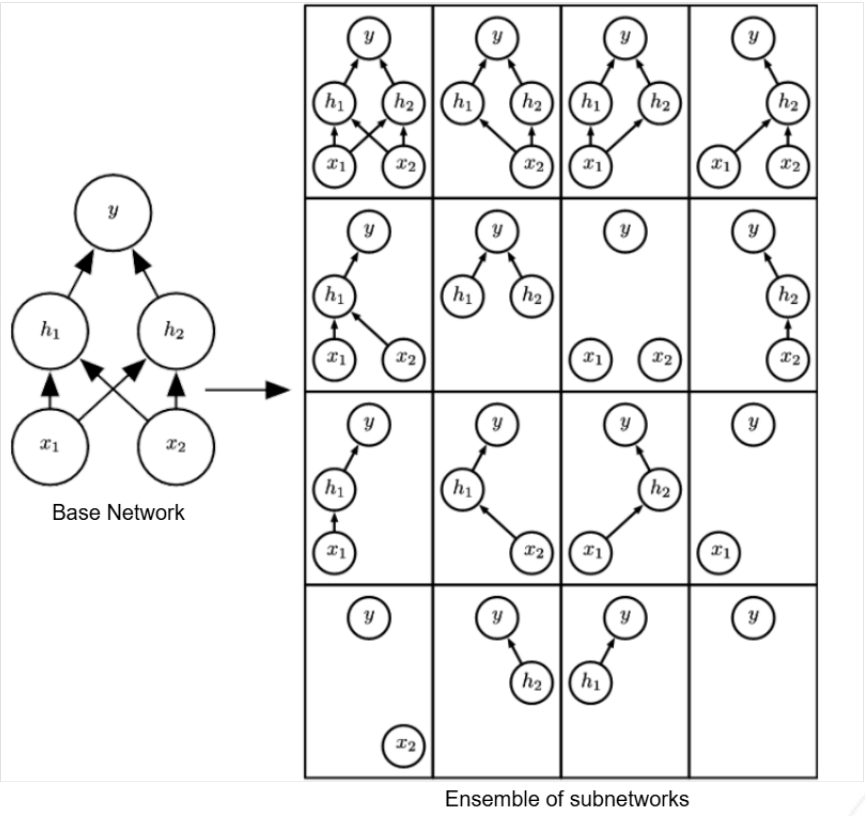


FIGURE 5.8: Dropout mechanism. Source: Lee et al. (2018).

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Figure 5.8 displays how dropout effectively creates an ensemble of subnetworks by randomly omitting different sets of neurons in each forward pass. In this example, the base network includes two input units and two hidden units, which can form up to 16 distinct subnetworks depending on which neurons are dropped. While smaller subnetworks may lose all input connections, this becomes increasingly rare in larger architectures with wider layers.

- **Early Stopping**

  When training large neural networks with significant representational power, models often exhibit a consistent pattern: while the training error decreases steadily over time, the validation error initially decreases but eventually starts to rise. This behavior indicates that the model has begun to overfit the training data.

  Early stopping helps prevent overfitting by stopping the training process at the point where the model performs best on the validation set. Instead of training until full convergence, the model is monitored, and the parameters corresponding to the lowest validation error are saved. Training is terminated when the validation error fails to improve for a predefined number of consecutive iterations, known as *patience*.

  This meta-algorithm identifies the optimal number of training steps to achieve the best generalization performance.

Algorithm 2 Early Stopping Algorithm

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let θ_0 be the initial parameters.
Initialize $\theta \leftarrow \theta_0$, $i \leftarrow 0$, $j \leftarrow 0$, $v \leftarrow \infty$
Set $\theta^* \leftarrow \theta$, $i^* \leftarrow i$
while $j < p$ **do**
  Update θ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\theta)$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\theta^* \leftarrow \theta$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters θ^*, best number of training steps i^*

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Dataset Augmentation

One of the best ways to improve a model's performance on unseen data, applicable not only to neural networks but to machine learning in general is to train it on a large and diverse dataset. However, in practice, collecting sufficient data is often difficult. This is where data augmentation becomes valuable: it creates additional training samples by applying transformations to existing data. These transformations increase the variability of the training set, helping the model learn more robust patterns and generalize better to new inputs. It is essential, however, that data augmentation is applied only to the training data. The evaluation (validation or test) dataset must remain completely independent and untouched by augmentation, to ensure that performance metrics reflect the model's ability to generalize to truly unseen data.

For time series data, augmentation techniques must ensure that the sequential relationships and inherent patterns remain intact. One widely used method is *jittering* which introduces small random noise to the data points, mimicking natural variations and improving the model's robustness to fluctuations. Another approach is *time warping* that stretches or compresses the time intervals between observations, allowing the model to adapt to temporal distortions.

A commonly applied technique is *window slicing* where random segments or windows of the time series are extracted. This helps the model learn from different parts of the sequence and improves its ability to recognize patterns across varying time scales. Additional transformations, such as *magnitude warping* which alters amplitude variations and *permutation-based methods* which shuffle specific segments while preserving the overall trend, further improve dataset variability.

Batch Normalization

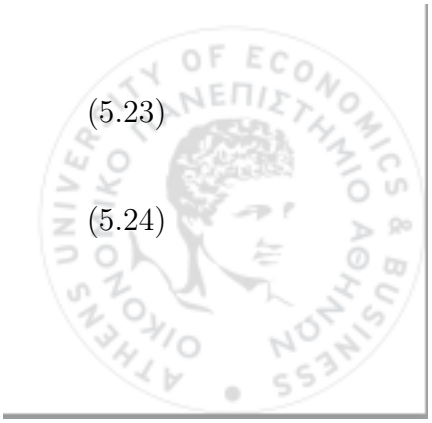
While techniques such as dropout and early stopping explicitly reduce overfitting by limiting model complexity, another technique called batch normalization provides an indirect form of regularization.

Batch normalization, introduced by Ioffe and Szegedy (2015), addresses internal covariate shift where the distribution of activations inside a neural network changes dynamically during training, slowing down convergence and making optimization more challenging. By normalizing activations to have a mean of zero and a variance of one, batch normalization reduces sensitivity to initialization and allows the use of higher learning rates, thereby accelerating training.

Given a mini-batch of activations $x_1, x_2, \ldots, x_m$, batch normalization computes the batch mean μ_B and variance σ_B^2 as:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{5.23}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{5.24}$$



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Each activation is then normalized:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{5.25}$$

where ϵ is a small constant added for numerical stability. To retain the network's expressiveness, two trainable parameters, γ (scale) and β (shift), are introduced:

$$y_i = \gamma \hat{x}_i + \beta \tag{5.26}$$

During training, the mean and variance are computed dynamically for each mini-batch. However, at test time, it uses a moving average of these statistics collected during training.

Despite its benefits, batch normalization is not a foolproof solution to overfitting. A model can still overfit if it is too complex relative to the available training data, if the dataset contains excessive noise or if the training process is not well-regularized. Therefore, it is crucial to combine batch normalization with other regularization techniques such as the one mentioned above, while carefully monitoring validation performance.

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Chapter 6

Neural Networks Strategies for Time Series Forecasting

In the previous chapter, a general analysis of neural networks and their underlying mechanisms was provided. With that understanding in place, the discussion now turns to specific architectures.

6.1 Recurrent Neural Networks

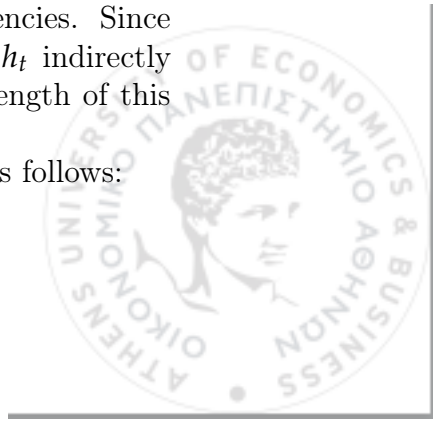
Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data by maintaining a form of memory through recurrent connections. Unlike traditional neural networks that assume each input is independent, RNNs use information from previous time steps to influence current predictions. This makes them particularly effective for tasks that involve temporal dependencies such as time series forecasting, natural language processing and speech recognition.

6.1.1 Simple Recurrent Network

The Simple Recurrent Network (SRN) was introduced by Elman (1990) in his paper titled "Finding Structure in Time". SRN is one of the earliest and simplest forms of RNNs. Unlike traditional feedforward networks such as MLPs, which require fixed-size input and lack memory of prior inputs, SRNs can handle variable-length sequences and retain information from previous inputs. This makes them particularly well-suited for modeling time series and other sequential data.

SRNs work by using a hidden state to carry information from one time step to the next, allowing them to process sequential data more effectively. They can be seen as a sequence of plain vanilla feedforward networks that share weights across time steps, where at each step the network updates its internal representation h_t based on both the current input x_t and its previous state h_{t-1}, allowing it to capture temporal dependencies. Since h_{t-1} is a function of both x_{t-1} and h_{t-2}, the hidden state h_t indirectly reflects information from earlier time steps, although the strength of this memory diminishes over time.

The forward propagation equations in an SRN are given as follows:



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

$$h_t = \tanh(Wh_{t-1} + Ux_t + b) \tag{6.1}$$

$$y_t = Vh_t + c \tag{6.2}$$

where:

- $h_t \in \mathbb{R}^n$ denotes the hidden state at time step t, where n is the hidden dimension (a hyperparameter). The initial state h_0 is commonly initialized to zero.
- $x_t \in \mathbb{R}^m$ is the input of a time series at time t, where m is the number of features at each time step. For a univariate time series, this would be a scalar ($x_t \in \mathbb{R}$).
- $y_t \in \mathbb{R}^k$ denotes the predicted output at time step t. In regression tasks, k matches the dimensionality of the target variable, while in classification tasks k corresponds to the number of classes. For univariate time series, y_t reduces to a scalar ($y_t \in \mathbb{R}$), commonly representing the predicted value of the series at $t+1$.
- $W \in \mathbb{R}^{n \times n}$ is the hidden-to-hidden transition matrix, mapping the previous hidden state h_{t-1} to the current hidden state h_t.
- $U \in \mathbb{R}^{n \times m}$ is the input-to-hidden transition matrix, mapping the input x_t to the current hidden state h_t.
- $V \in \mathbb{R}^{k \times n}$ is the hidden-to-output transformation matrix, mapping the hidden state h_t to the output y_t.
- $b \in \mathbb{R}^n$ and $c \in \mathbb{R}^k$ are bias terms associated with the hidden and output layers respectively.
- $\tanh(\cdot)$ is the activation function applied to the hidden state, making sure values remain bounded within the range $(-1, 1)$.

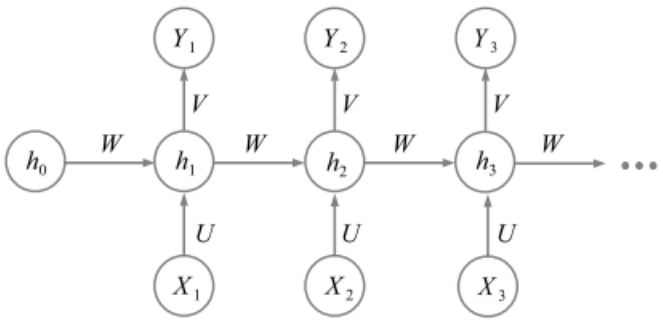


FIGURE 6.1: The architecture of a Simple Recurrent Network.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

The hidden state h_t acts like a memory, continuously updating itself based on the previous state h_{t-1} and the new input x_t. This allows the model to track trends and seasonality over time. The output y_t is computed as a transformation of the hidden state, where the matrices W, U, V and biases b, c are adjusted during training (via backpropagation) to optimize the final prediction. Different activation functions are used depending on the task such as softmax for classification and a linear function for continuous predictions. Figure 6.1 illustrates the architecture of an SRN.

Loss Functions

Defining a loss function is essential for measuring the difference between predicted and actual outputs in recurrent networks. This function helps to measure and guide model optimization.

In a recurrent network, the loss function sums up errors from all time steps:

$$L = \sum_{t=1}^{T} L_t \tag{6.3}$$

where L_t represents the individual loss at time step t. Some common choices for the individual loss can be the following, where Y_t and Z_t are the model's prediction and the target value at time step t respectively:

1. Mean Squared Error (MSE):

$$L_t = \mathbb{E}\left[|Y_t - Z_t|^2\right]. \tag{6.4}$$

2. Kullback-Leibler Divergence (KL Divergence):

$$L_t = D_{KL}\left(p_{(X_1,\ldots,X_t),Z_t} \parallel p_{\theta;(X_1,\ldots,X_t),Z_t}\right). \tag{6.5}$$

3. Cross-Entropy Loss:

$$L_t = -\mathbb{E}^{p_{(X_1,\ldots,X_t),Z_t}}\left[\log p_{\theta;(X_1,\ldots,X_t),Z_t}\right]. \tag{6.6}$$

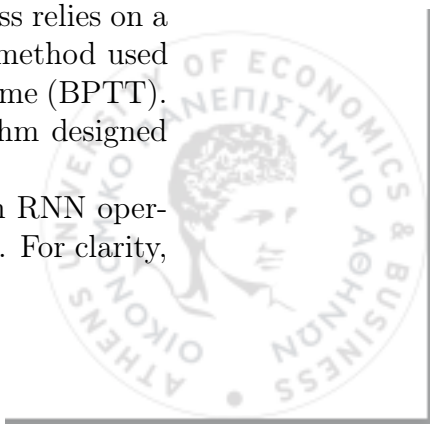
4. Squared Euclidean Distance:

$$L_t = \frac{1}{2}(Y_t - Z_t)^2. \tag{6.7}$$

Backpropagation Through Time

In an SRN, the forward pass updates hidden states h_t, generates outputs Y_t and computes the loss L_t at each time step. Minimizing this loss relies on a gradient-based approach that requires computing $\nabla_\theta L$. The method used for this computation is known as Backpropagation Through Time (BPTT). BPTT is a specialized version of the backpropagation algorithm designed to train RNNs.

To simplify the explanation of BPTT, an example with an RNN operating over three time steps, as illustrated in Figure 6.2, is used. For clarity,



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

all variables are assumed to be scalars. This assumption makes it easier to illustrate the core mechanics of BPTT without involving vector or matrix calculus. In practice, these quantities are typically vectors or matrices, and the corresponding derivatives require Jacobians and matrix operations.

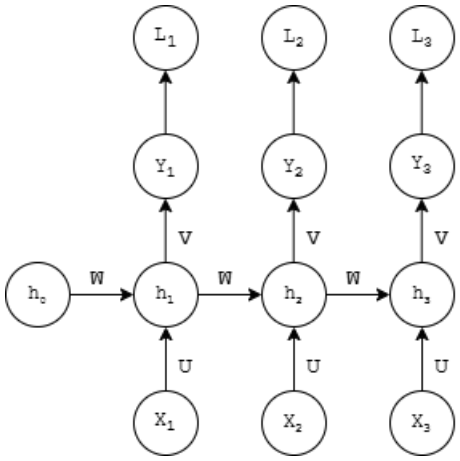


FIGURE 6.2: Example of an RNN.

The forward propagation equations are:

$$a_1 = Wh_0 + UX_1 + b, \quad a_2 = Wh_1 + UX_2 + b, \quad a_3 = Wh_2 + UX_3 + b$$
$$h_1 = \tanh(a_1), \quad h_2 = \tanh(a_2), \quad h_3 = \tanh(a_3)$$
$$Y_1 = Vh_1 + c, \quad Y_2 = Vh_2 + c, \quad Y_3 = Vh_3 + c$$

For simplicity, the loss function is defined as:

$$L = L_1 + L_2 + L_3 = \frac{1}{2}(Y_1 - Z_1)^2 + \frac{1}{2}(Y_2 - Z_2)^2 + \frac{1}{2}(Y_3 - Z_3)^2$$

The following gradient components must be computed:

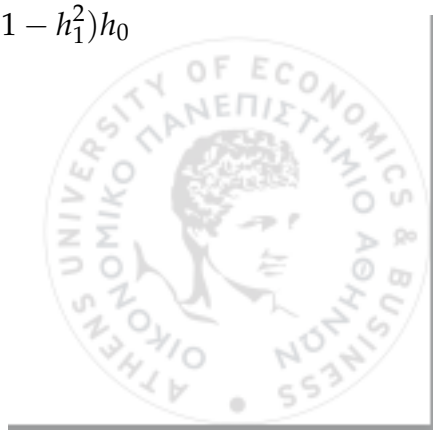
$$\nabla_\theta L = (\frac{\partial L}{\partial W}, \frac{\partial L}{\partial V}, \frac{\partial L}{\partial U}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial c})$$

Using the chain rule:

$$\frac{\partial h_1}{\partial W} = \frac{\partial}{\partial W}\tanh(a_1) = \operatorname{sech}^2 a_1 \frac{\partial a_1}{\partial W} = (1 - \tanh^2 a_1)h_0 = (1 - h_1^2)h_0$$

Respectively,

$$\frac{\partial h_2}{\partial W} = (1 - h_2^2)h_1, \quad \frac{\partial h_3}{\partial W} = (1 - h_3^2)h_2$$



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

$$\frac{\partial h_1}{\partial b} = 1 - h_1^2, \quad \frac{\partial h_2}{\partial b} = 1 - h_2^2, \quad \frac{\partial h_3}{\partial b} = 1 - h_3^2$$

Furthermore,

$$\frac{\partial L_1}{\partial h_1} = \frac{\partial L_1}{\partial Y_1}\frac{\partial Y_1}{\partial h_1} = (Y_1 - Z_1)V, \quad \frac{\partial L_2}{\partial h_2} = (Y_2 - Z_2)V, \quad \frac{\partial L_3}{\partial h_3} = (Y_3 - Z_3)V$$

Applying the chain rule again:

$$\frac{\partial h_1}{\partial U} = \frac{\partial}{\partial U}\tanh a_1 = (1 - h_1^2)\frac{\partial a_1}{\partial U} = (1 - h_1^2)X_1,$$
$$\frac{\partial h_2}{\partial U} = (1 - h_2^2)X_2, \quad \frac{\partial h_3}{\partial U} = (1 - h_3^2)X_3$$

Lastly,

$$\frac{\partial h_2}{\partial h_1} = \frac{\partial}{\partial h_1}\tanh(a_2) = \operatorname{sech}^2(a_2)\frac{\partial a_2}{\partial h_1} = (1 - \tanh^2(a_2))W = (1 - h_2^2)W$$

$$\frac{\partial h_3}{\partial h_2} = (1 - h_3^2)W, \quad \frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1} = (1 - h_3^2)(1 - h_2^2)W^2$$

The gradient is computed as follows:

$$\frac{\partial L}{\partial V} = \frac{\partial L_1}{\partial V} + \frac{\partial L_2}{\partial V} + \frac{\partial L_3}{\partial V} = \cdots = \sum_t (Y_t - Z_t)h_t$$

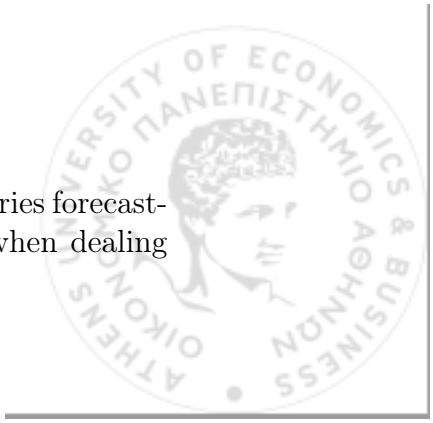
$$\frac{\partial L}{\partial c} = \frac{\partial L_1}{\partial c} + \frac{\partial L_2}{\partial c} = \frac{\partial L_1}{\partial Y_1}\frac{\partial Y_1}{\partial c} + \frac{\partial L_2}{\partial Y_2}\frac{\partial Y_2}{\partial c} = \cdots = \sum_t (Y_t - Z_t)$$

$$\begin{aligned}\frac{\partial L}{\partial W} = \frac{\partial L_1}{\partial W} + \frac{\partial L_2}{\partial W} + \frac{\partial L_3}{\partial W} = \cdots &= (Y_1 - Z_1)V(1 - h_1^2)h_0 \\ &+ (Y_2 - Z_2)V(1 - h_2^2)h_1 \\ &+ (Y_2 - Z_2)V(1 - h_2^2)W(1 - h_1^2)h_0 \\ &+ (Y_3 - Z_3)V(1 - h_3^2)h_2 \\ &+ (Y_3 - Z_3)V(1 - h_3^2)W(1 - h_2^2)h_1 \\ &+ (Y_3 - Z_3)V(1 - h_3^2)W(1 - h_2^2)W(1 - h_1^2)h_0\end{aligned}$$

Respectively for $\frac{\partial L}{\partial U}$ and $\frac{\partial L}{\partial b}$.

Limitations of Recurrent Neural Networks

RNNs have been widely used for sequence modeling and time series forecasting. However, they exhibit several limitations, particularly when dealing



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

with long sequences. One of the most critical issues in training RNNs is the vanishing gradient problem, which occurs during BPTT. In the gradient computations in the three-step RNN example, were observed that the gradients with respect to W, U, and b involve multiple products containing the matrix W, as well as the activation derivatives $(1 - h_t^2)$. Given $h_t = \tanh(a_t) \in (-1, 1)$, it follows that $1 - h_t^2 \in (0, 1)$. Since each gradient expression contains factors of the form $(1 - h_t^2)$, the gradients tend to decrease as they propagate backward through time. The longer the sequence, the more these small multiplicative terms accumulate, making the gradient vanish exponentially.

In the case of an RNN spanning multiple time steps, the BPTT involves taking derivatives of the form: $\frac{\partial h_t}{\partial h_{t-1}}$ which means that for an earlier time step, the gradient accumulates multiple products:

$$\frac{\partial h_T}{\partial h_1} = \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} = (1 - h_T^2)W(1 - h_{T-1}^2)W \dots (1 - h_2^2)W$$

Rewriting in terms of powers of W:

$$\frac{\partial h_T}{\partial h_1} = W^{T-1} \prod_{t=2}^{T} (1 - h_t^2)$$

If the eigenvalues of W satisfy $|\lambda_i| < 1$:

$$|\lambda_i|^{T-1} \to 0 \quad \text{as} \quad T \to \infty.$$

causing the gradient to vanish exponentially. This confirms that the vanishing gradient problem becomes more severe as the sequence length increases.

Conversely, if the matrix W has eigenvalues $|\lambda_i| > 1$, then:

$$|\lambda_i|^{T-1} \to \infty \quad \text{as} \quad T \to \infty.$$

This results in the exploding gradient problem where gradients grow exponentially and make training unstable.

As an example of the vanishing gradient problem, an RNN trained to answer the question: "What time is it?" is considered.

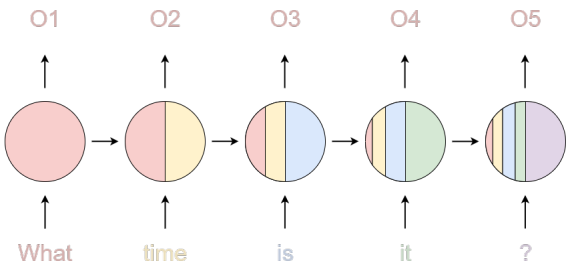


FIGURE 6.3: The vanishing gradient problem.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

At each time step, the network processes one word while passing its hidden state to the next time step. Ideally, when reaching the last word, the network should retain enough context from the earlier words to generate a meaningful response. However, due to the vanishing gradient problem, information from the earlier words ("What" and "time") weakens as it propagates through the network. By the time the RNN reaches the final step ("?"), most of the context have faded (Figure 6.3). This forgetting effect becomes even more obvious as the sequence length increases. The deeper the network in terms of time steps, the more significant the loss of past information.

6.1.2 Long Short-Term Memory Network

Long Short-Term Memory (LSTM) networks were introduced by Hochreiter and Schmidhuber (1997) as a solution to the vanishing gradient problem that affects standard RNNs. This is a type of RNN that includes special cells designed to learn long-term dependencies. LSTMs use a gating mechanism that selectively regulates the flow of information through a combination of a sigmoid layer and pointwise multiplication. Their functionality revolves around three gates: forget, input and output (Figure 6.4). At the core of an LSTM is the internal cell state C_t, which enables the network to maintain long-term information efficiently, and an internal loop within each cell. The roles of each gate are described as follows:

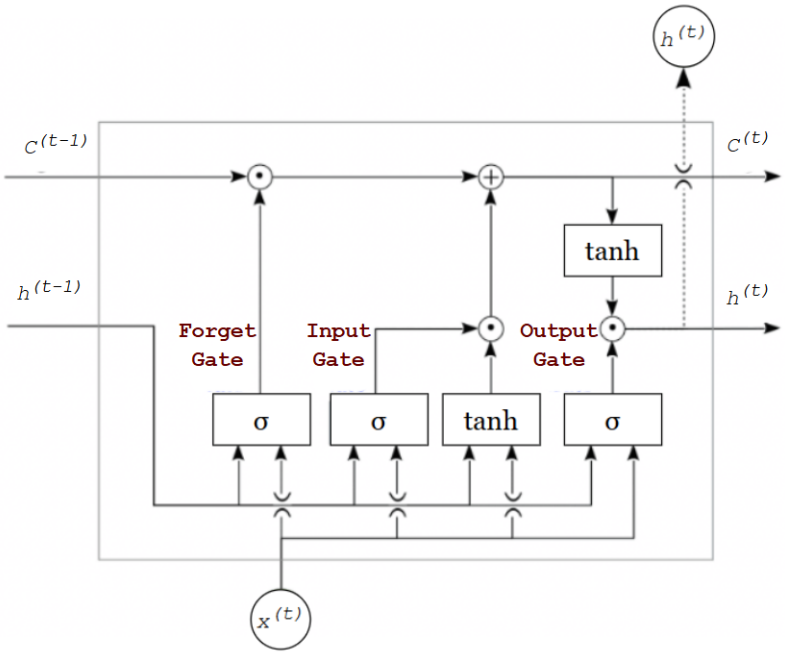


FIGURE 6.4: A standard Long Short-Term Memory architecture.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

1. **The forget gate**

   This gate is a sigmoid layer responsible for determining which portions of previous data should be retained or discarded, making sure that the model focuses on relevant historical patterns.

   It is defined as:

$$f_t = \sigma(W_f h_{t-1} + U_f X_t + b_f) \tag{6.8}$$

   where: $W_f \in \mathbb{R}^{n \times n}$ and $U_f \in \mathbb{R}^{n \times m}$ are trainable matrices, $b_f \in \mathbb{R}^n$ is a trainable bias vector and $X_t \in \mathbb{R}^m$ is the input of a time series at time step t. Since σ outputs values between 0 and 1, f_t is a vector that determines the proportion of past information that should be retained or discarded. If the values are closer to 0, the more the corresponding past information is ignored. In contrast, values close to 1 indicate that the information should be preserved for future processing.

2. **The input gate**

   It is responsible for adjusting the internal cell state by integrating new information while preserving relevant past data. It is expressed as:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{6.9}$$

   where $\odot$ is the element-wise multiplication operator and i_t is a scaling factor with formula:

$$i_t = \sigma(W_i h_{t-1} + U_i X_t + b_i) \tag{6.10}$$

   The candidate cell state $\tilde{C}_t$ represents the new information that could be included and is computed as:

$$\tilde{C}_t = \tanh(W_c h_{t-1} + U_c X_t + b_c) \tag{6.11}$$

   Since the hyperbolic tangent function outputs values within the range (-1,1), it ensures the candidate contribution remains within this interval. The product $i_t \tilde{C}_t$ controls how much of the new candidate information is added to the state, while $f_t C_{t-1}$ determines how much of the previous state is retained after filtering through the forget gate.

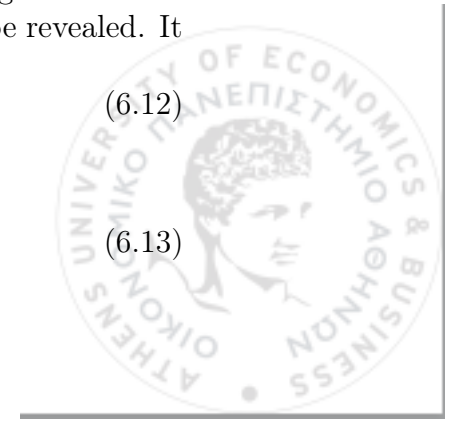
3. **The output gate**

   This gate determines the hidden state h_t by controlling the amount of information from the internal cell state that should be revealed. It is defined as:

$$h_t = o_t \odot \tanh(C_t) \tag{6.12}$$

   where o_t is a gating factor with formula:

$$o_t = \sigma(W_o h_{t-1} + U_o X_t + b_o) \tag{6.13}$$



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

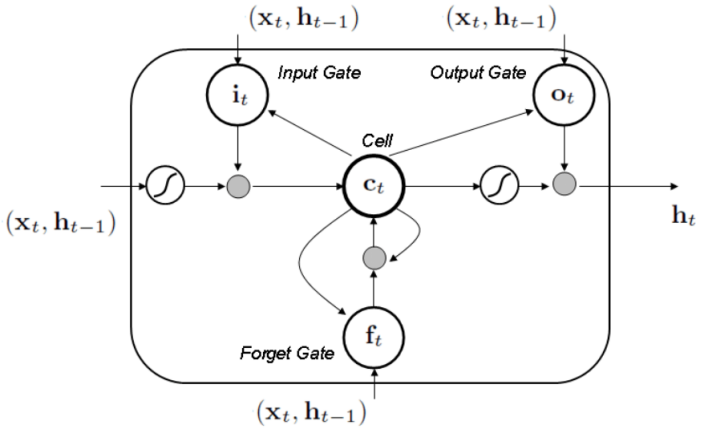


FIGURE 6.5: Long Short-Term Memory with peephole connections. Source: Fernandez et al. (2014).

LSTMs are not always implemented in the exact same way as described above. In fact, nearly every research paper that uses LSTMs appears to adopt a slightly different variation of the architecture. A well-known variation of LSTMs, introduced by Gers and Schmidhuber (2000), is adding *peephole connections*. This approach allows the gate layers to directly access information from the cell state. In Figure 6.5, peephole connections are added into all the gates. However, different research studies adopt a selective approach, integrating peepholes into specific gates while excluding them from others.

6.1.3 Gated Recurrent Unit Network

Like LSTMs, Gated Recurrent Unit (GRU) Networks are designed to handle time series by selectively retaining or discarding information over time. However, GRUs offer a simpler architecture (Figure 6.6) with fewer parameters which can make them more efficient and easier to train. Introduced by Cho et al. (2014), GRUs use two gating mechanisms, the update gate and the reset gate, to regulate information flow.

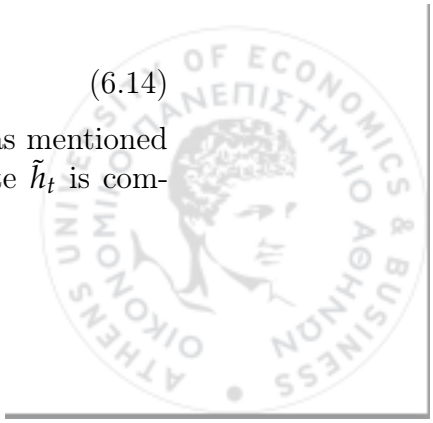
1. **The reset gate**

   It determines how much of the previous hidden state should be retained when computing the new candidate hidden state. This allows the model to selectively forget past information.

   It is defined as:

$$r_t = \sigma(W_r h_{t-1} + U_r X_t + b_r) \tag{6.14}$$

   where the weights and biases follow the same notation as mentioned before. Using the reset gate, the candidate hidden state $\tilde{h}_t$ is computed as:



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

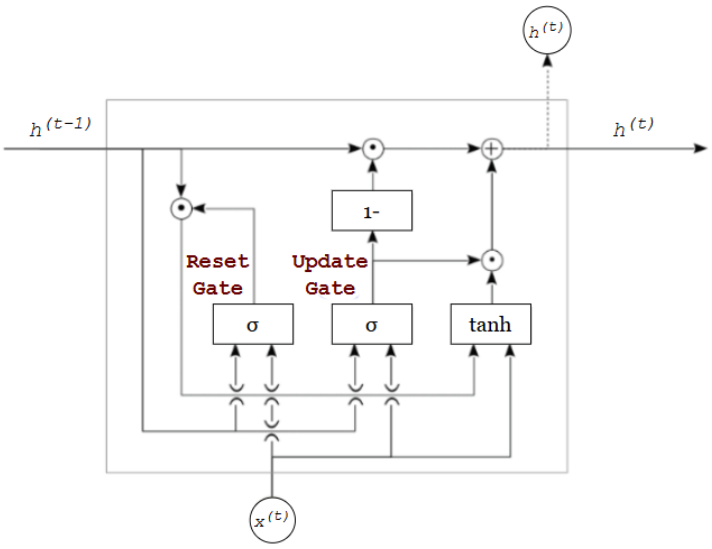


FIGURE 6.6: Gated Recurrent Unit architecture.

$$\tilde{h}_t = \tanh(W_h(r_t \odot h_{t-1}) + U_h X_t + b_h) \tag{6.15}$$

2. **The update gate**

   This gate controls how much of the previous hidden state should be carried forward to the next time step, regulating the balance between past and new information.

   The update gate is computed as:

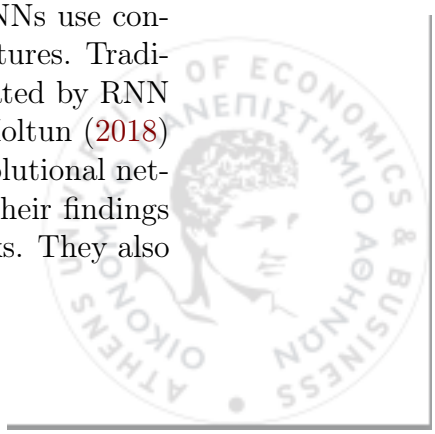
$$u_t = \sigma(W_u h_{t-1} + U_u X_t + b_u) \tag{6.16}$$

   The final hidden state h_t is then updated as:

$$h_t = u_t \odot h_{t-1} + (1 - u_t) \odot \tilde{h}_t \tag{6.17}$$

6.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of artificial neural networks designed to process structured data such as images and sequential signals. Inspired by neurons in human and animal brains, CNNs use convolutional layers to extract patterns as well as hierarchical features. Traditionally, sequence modeling in deep learning has been dominated by RNN architectures. However, recent research by Bai, Kolter, and Koltun (2018) challenges this conventional approach and suggests that convolutional networks are strong alternatives for handling sequential data. Their findings demonstrate that CNNs can outperform RNNs in various tasks. They also



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

effectively reduce common issues such as the vanishing or exploding gradient problem. Moreover, CNNs enable parallel computation of outputs which leads to significant performance improvements over recurrent models.

For the purposes of this thesis, the focus will be on one-dimensional CNNs (1D-CNNs) as well as Temporal Convolutional Networks (TCNs), both of which have proven to be highly effective in processing sequential data, including time series. The network architecture of a TCN builds upon the principles of a 1D CNN, with a series of 1D convolutional layers stacked together. Although the primary focus is on their application to time series data, understanding CNN architectures and how they work is essential.

6.2.1 Structure of Convolutional Neural Networks

The CNN architecture includes multiple layers, with three primary types explained in detail.

1. **Convolutional Layer**

   A convolutional layer consists of a set of learnable filters, also known as kernels, that are applied to the input data to extract meaningful patterns. Each filter has three dimensions: height, width, and depth. The height and width are relatively small spatially, while the depth extends through the entire depth of the input volume. This means the filter's depth matches the number of channels in the input data. For example, a grayscale image has a depth of 1, while a color image, which contains RGB channels, has a depth of 3.

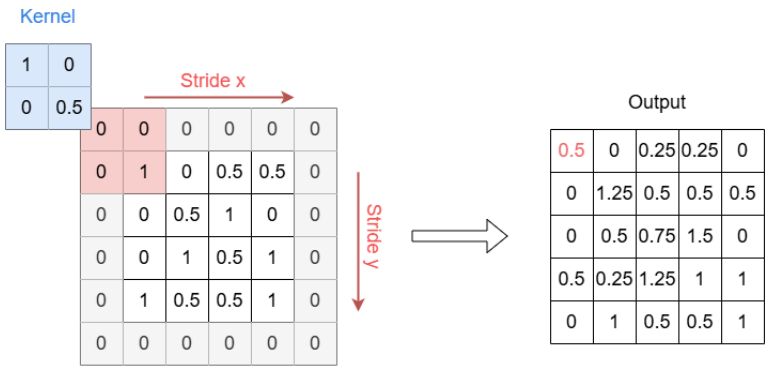


FIGURE 6.7: Visualization of 2D convolution process.

During the forward pass, each filter slides over the input volume along its width and height, performing a convolution operation. At each

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

position, the filter performs element-wise multiplication with the corresponding region of the input, and the resulting values are summed to produce a single output value. This process is repeated across the entire input volume. Figure 6.7 shows this process, where the filter moves across the input, performing element-wise multiplication at each position to generate the output feature map.

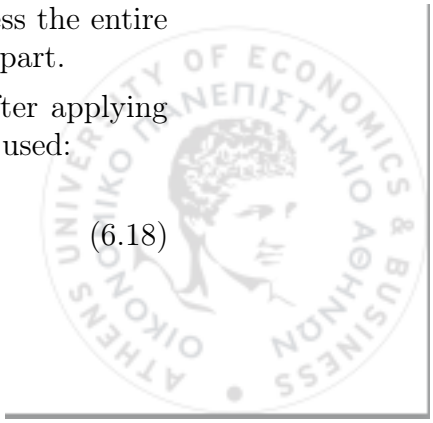
After the convolution, a nonlinear activation function is applied to add non-linearity. The resulting output is known as a feature map or activation map, which represents the filter's response to specific patterns or features detected in different regions of the input.

The dimensions of the output volume in a convolutional layer are determined by three main hyperparameters: depth, stride, and padding.

- **Depth**
  This refers to the total number of filters used during the convolution process. Each filter is responsible for identifying different features within the input such as edges, textures or specific patterns. As the number of filters increases, the output volume becomes deeper, with each filter contributing its own feature map.
- **Stride**
  The stride defines how far the filter moves across the input image with each step. A stride of 1 shifts the filter one pixel at a time, while a stride of 2 causes it to jump two pixels per step. Increasing the stride reduces the spatial dimensions of the output, as fewer positions are covered during the convolution.
- **Padding**
  Padding is the technique of adding extra zero value pixels around the input's border. This helps manage the output dimensions and prevents significant reduction in size after convolution. Padding comes in three common forms:
  (a) *Valid Padding*
  No padding is applied which leads to a smaller output compared to the input dimensions.
  (b) *Same Padding*
  Padding is added in such a way that the output dimensions remain the same as the input.
  (c) *Full Padding*
  With full padding, the filter can reach all the way to the edges of the input. This way, it's able to process the entire input from start to finish without missing any part.

  To calculate the size of the output feature map after applying the convolution operation, the following formula is used:

$$O = \frac{I - F + P_{\text{start}} + P_{\text{end}}}{S} + 1 \tag{6.18}$$



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Where:

- O is the output size,
- I is the input size,
- F is the filter size,
- P_{start} and P_{end} is the padding added to the beginning and end of the input,
- S is the stride

Frequently, $P_{\text{start}} = P_{\text{end}} \stackrel{\triangle}{=} P$ which allows substituting $P_{\text{start}} + P_{\text{end}}$ with $2P$ in the formula above.

2. **Pooling Layer**

The pooling layer is a downsampling technique commonly applied after convolutional layers. Its primary role is to reduce the spatial dimensions of feature maps. There are two widely used types of pooling operations:

(a) *Max Pooling*
It selects the highest value within each kernel, focusing on the most important features in the data. As a result, it is highly effective for tasks such as object detection.

(b) *Average Pooling*
It calculates the average of all values within the kernel and smooths the output by reducing the impact of extreme values. Thus, it provides a more generalized representation.

Figure 6.8 displays the difference between max pooling and average pooling. On the left, max pooling selects the maximum value from each region covered by the kernel, while on the right, average pooling computes the mean of values within each region.

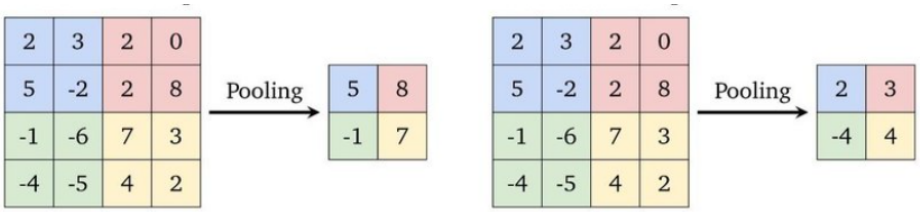


FIGURE 6.8: Comparison of max pooling (left) and average pooling (right) operations.

3. **Fully Connected Layer**

The fully connected layer works with a flattened input where every input node is connected to all neurons in the layer (Figure 6.9). This layer is usually placed towards the end of CNN architectures.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

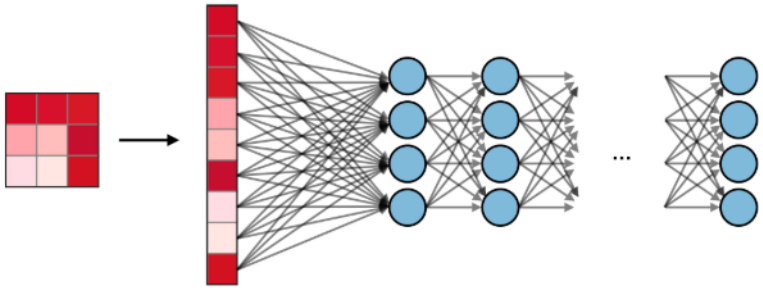


FIGURE 6.9: Structure of a fully connected layer. Source: Altun et al. (2023).

6.2.2 One-dimensional Convolutional Neural Networks

The CNNs discussed earlier are specifically designed to process two-dimensional data, such as images and videos, which is why they are commonly referred to as *2D CNNs*. As an alternative, a modified version known as 1D CNNs has been developed. 1D CNNs are often preferred when working with sequential data because of their computational efficiency, as they rely on sequences of 1D convolutions that involve simpler filter operations with fewer parameters and lower memory requirements compared to 2D CNNs. Moreover, they have the ability to capture temporal dependencies, which makes them suitable for not only one-dimensional signals, such as univariate time series data, but also multivariate time series data where multiple variables are observed over time.

A 1D convolutional layer takes as input a three-dimensional tensor and produces an output that is also a three-dimensional tensor. The dimensions are structured as follows:

- Input Tensor Shape: (batch size, input length, input channels)
- Output Tensor Shape: (batch size, output length, output channels)

Despite both the input and output being 3D tensors, the layer is referred to as a 1D convolutional layer because the convolution operation is performed along only one dimension. The convolution filters slide across the time axis to capture temporal dependencies.

In the case of univariate time series data, there is only one feature observed over time, which means that the input channels equal 1. Similarly, the output channels can also be set to 1 if only one filter is applied, though multiple filters can be used to capture different patterns in the data. In Figure 6.10, the convolution operation slides a kernel across the time axis. The kernel has a fixed size (e.g., 3 time steps) and performs a dot product between the kernel weights and the corresponding input values within each window. After each computation, the kernel shifts one step to the right (or

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

more, depending on the specified stride), applying the same set of weights across the entire sequence.

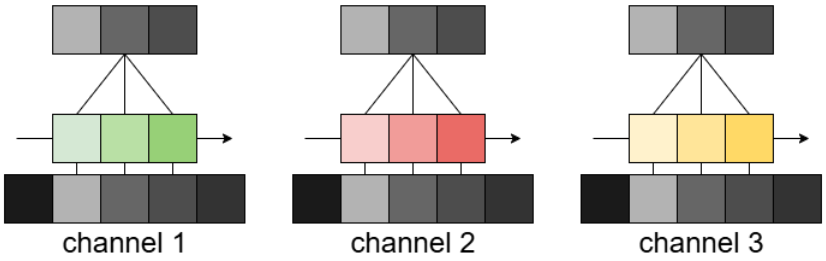


FIGURE 6.10: Convolution operation in univariate time series data.

For multivariate time series data, multiple features are observed simultaneously over time, which means that the input channels are greater than 1, with each channel representing a distinct feature. Similarly, the output channels can be set to 1 if a single filter is applied.

In Figure 6.11, the convolution operation slides a kernel along the time axis, just as in the univariate case. However, in this case, the kernel covers all input channels at once, performing element-wise multiplications between the kernel weights and the corresponding input values from each feature within the selected time window. The results of these multiplications are then summed together to produce a single output value for that specific position in the sequence.

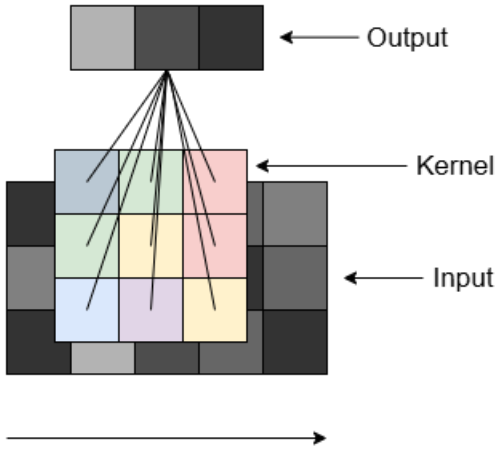


FIGURE 6.11: Convolution operation in multivariate time series data.

Although the primary focus is on univariate time series, mentioning multivariate time series provides context for how convolutional operations

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

can be extended to multiple features. This distinction helps illustrate the broader applicability of convolutional techniques and how they may be adapted for more complex time series data in future work.

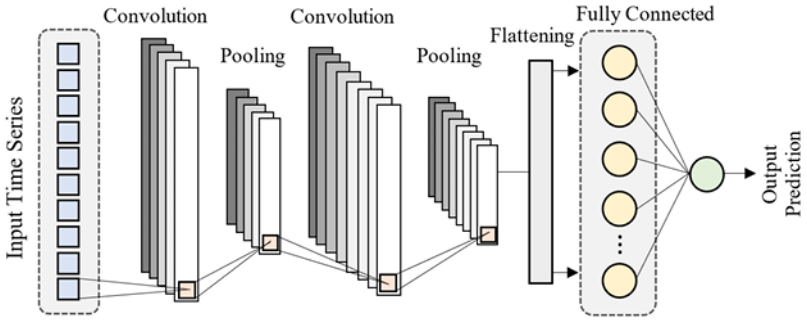


FIGURE 6.12: One-dimensional Convolutional Neural Network architecture. Source: Rahman et al. (2021).

Figure 6.12 displays the architecture of a 1D CNN designed for processing univariate time series data. The layers in this architecture operate in the same way as previously described. The only difference is the inclusion of the flatten layer, which acts as a bridge between the feature extraction layers (convolutional and pooling layers) and the final prediction stage (fully connected layer). After the convolution and pooling operations, the data exist as a 3D tensor. The flatten layer transforms this multi-dimensional output into a one-dimensional vector for each sequence in the batch.

6.2.3 Convolution vs Cross-Correlation

In the context of CNNs, the term *convolution* is frequently used to describe an operation that, in practice, is technically closer to *cross-correlation*. This distinction is important to clarify, especially when presenting mathematical formulations in this thesis.

Mathematical Formulation of 1D Convolution

The true mathematical convolution of a one-dimensional input sequence x with a filter f is defined as:

$$F(x_t) = (x * f)(t) = \sum_{j=-k}^{k} f[j] \cdot x[t-j] \tag{6.19}$$

Where:

- x is the input sequence,
- f is the convolutional filter of size $2k+1$,

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

- t represents the current time step,
- j indicates the relative position within the filter window

In this formulation, the key characteristic of convolution is the reversal of the filter f before performing the element-wise multiplication with the input sequence x. This reversal aligns with the original mathematical definition of convolution used in signal processing.

Cross-Correlation in CNNs

However, in CNNs, the operation commonly referred to as *convolution* is actually *cross-correlation*, which is mathematically expressed as:

$$F(x_t) = (x \star f)(t) = \sum_{j=-k}^{k} f[j] \cdot x[t+j] \tag{6.20}$$

The primary difference here is the absence of the filter reversal. In cross-correlation:

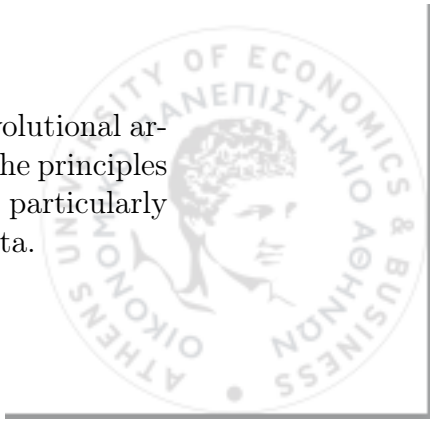
- The filter f is applied directly to the input sequence without flipping.
- The index $t+j$ indicates a straightforward sliding of the filter across the input.

While this distinction is mathematically significant, it has little practical impact on the performance of CNNs because the filter weights are learned during training. Thanks to the learning process, the network naturally adjusts for the missing filter reversal, so cross-correlation behaves much like convolution in most deep learning tasks.

Despite this practical equivalence, one might wonder why these models are still referred to as CNNs instead of the more technically accurate Cross-Correlation Neural Networks. The reason for this has to do with their history. When Yann LeCun introduced CNNs with LeNet in the late 1980s and early 1990s, the actual operation implemented was cross-correlation, not strict mathematical convolution (LeCun et al., 1989; LeCun et al., 1998). The term *convolution* was likely chosen because of its established use in signal processing and feature extraction. Over time, deep learning frameworks continued to use cross-correlation, as it avoids the unnecessary computational step of kernel flipping. Nonetheless, the name *Convolutional Neural Networks* persisted due to widespread recognition and adoption within the research community.

6.2.4 Temporal Convolutional Networks

Temporal Convolutional Networks (TCNs) are specialized convolutional architectures designed for sequence modeling tasks. Building on the principles of 1D CNNs, TCNs introduce modifications that make them particularly effective for capturing temporal dependencies in sequential data.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

- The convolutions in the architecture are causal, which means that data from future time steps do not affect or interfere with the processing of past information, thereby preserving the natural temporal sequence.
- The architecture is capable of processing input sequences of any length and producing output sequences of equal length, similar to the functionality of an RNN.

It is important to note that extended effective history ranges, referring to the network's capacity to retrieve information from distant past time steps for making predictions, are achieved through a combination of deep network architectures supported by residual layers and the use of dilated convolutions.

Causal Convolutions

A causal convolution ensures that the output at time t depends solely on inputs from the current and previous time steps, preventing any influence from future data. This operation is mathematically represented as:

$$F(x_t) = (x * f)(t) = \sum_{j=0}^{k-1} f[j] \cdot x[t-j] \tag{6.21}$$

where k is the size of the filter. To maintain the output sequence's length equivalent to the input sequence, zero-padding of length $k-1$ is applied to the beginning of the input.

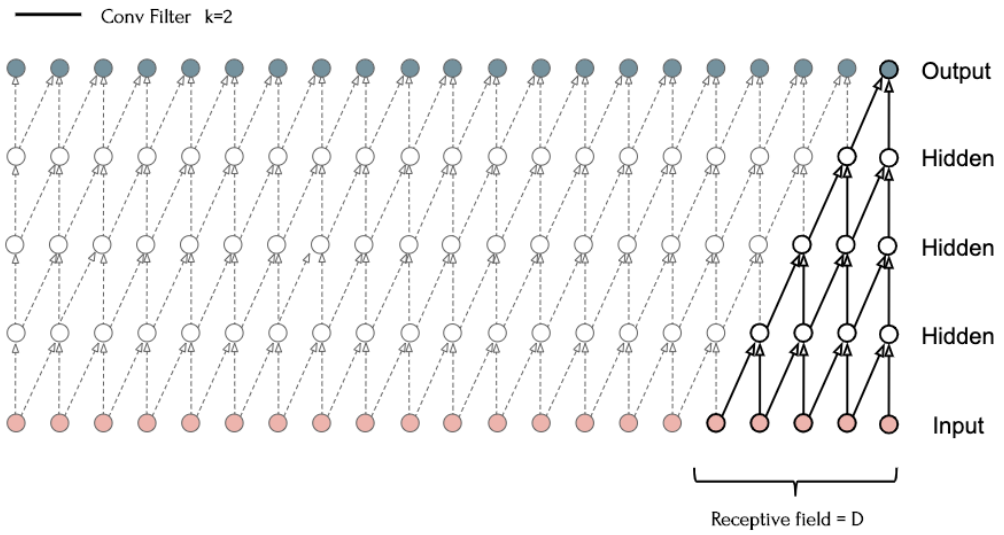


FIGURE 6.13: Stacked 1D convolutions.

Stacking multiple 1D convolutional layers results in a deeper architecture, where each output depends on an increasingly larger history of

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

input data. The range of input data that influences a particular output is referred to as the receptive field. It represents the number of previous time steps that contribute to the computation of the current output. For example, with four stacked convolutional layers and a filter size of $k = 2$, the receptive field covers inputs from five time steps (Figure 6.13). To address the need for larger receptive fields without excessively deep networks, dilated convolutions are used.

Dilated Convolutions

Dilated convolutions extend standard causal convolutions by introducing a dilation factor d, which determines the spacing between the elements of the convolution filter.

$$F(x_t) = (x *_d f)(t) = \sum_{j=0}^{k-1} f[j] \cdot x[t - (d \cdot j)] \tag{6.22}$$

Here, f is the filter of size k and x is the input sequence. When applying dilated convolutions across multiple layers, the dilation factor is typically exponentially increased. For the i-th layer, the dilation factor is $d_i = 2^{i-1}$. The receptive field for a 1D convolutional block with depth D and kernel size k is:

$$\text{Receptive Field} = 1 + (k - 1) \cdot (2^D - 1) \tag{6.23}$$

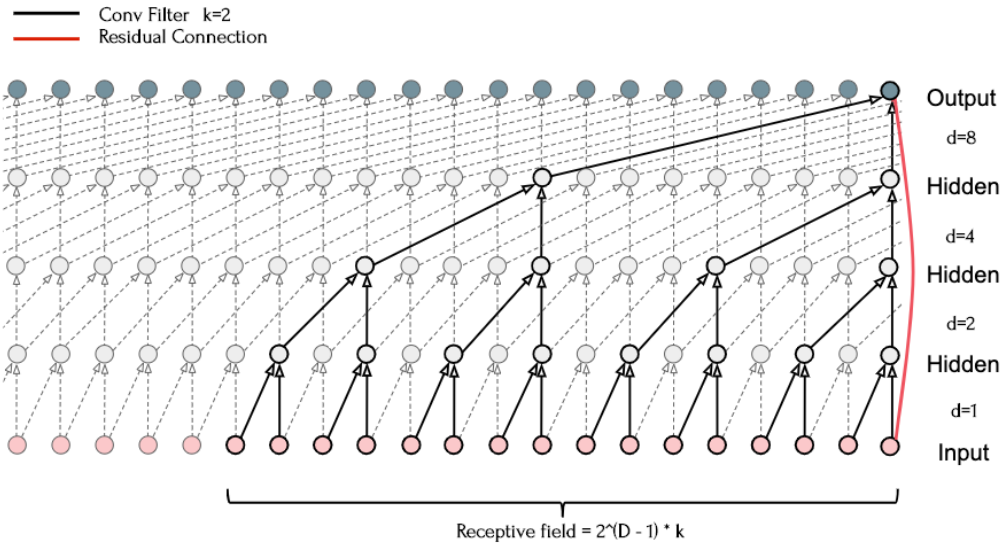


FIGURE 6.14: Dilated convolutions.

Figure 6.14 displays how dilated convolutions expand the receptive field through multiple layers. The dilation factor doubles at each layer

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

($d = 1, 2, 4, 8$). The red line represents a residual connection, allowing information to flow directly from the input to the output layer. The simplified expression: Receptive Field $= 2^{(D-1)} \cdot k$, is valid only if the kernel size is exactly 2.

Residual Connections

As neural networks deepen, propagating errors backward becomes challenging due to the vanishing gradient problem. Residual connections help reduce this by providing a direct path for gradients. This is achieved through a skip connection that allows the input to bypass one or more layers, thus preserving the flow of information. The output of a residual block is defined as:

$$o = \text{Activation}(x + F(x)) \tag{6.24}$$

where o is the output of the residual block, x is the original input to the block and $F(x)$ represents a series of transformation functions applied to the input. Here, the transformations F correspond to the computations carried out within the 1D convolutional layers. Figure 6.15 displays the residual block for TCN.

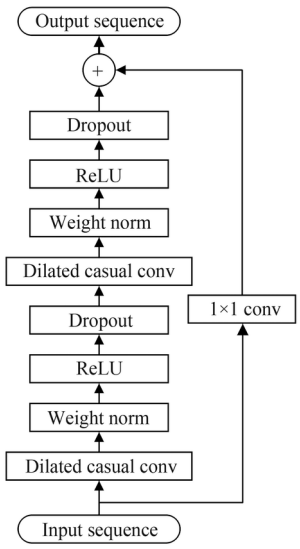
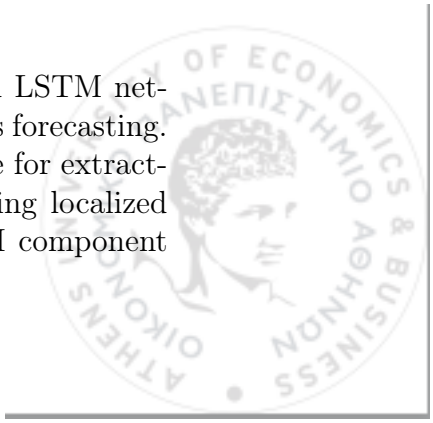


FIGURE 6.15: The residual block in TCN architecture.

6.2.5 CNN-LSTM

The CNN-LSTM model combines the strengths of CNNs and LSTM networks, making it particularly effective for univariate time series forecasting. In this hybrid architecture, the CNN component is responsible for extracting short-term patterns from the input sequence by identifying localized dependencies and fluctuations. At the same time, the LSTM component



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

models long-term temporal relationships, which enables the network to recognize overarching trends. This approach has been widely applied to various time series forecasting tasks. Livieris, Pintelas, and Pintelas (2020) introduced a CNN-LSTM model for gold price forecasting and demonstrated its effectiveness in capturing price movements and improving predictive accuracy. In a similar study, Alhussein, Khursheed, and Haider (2020) applied a CNN-LSTM model to short-term individual household load forecasting and showed that the model successfully learned electricity consumption patterns.

Figure 6.16 presents a visual representation of the CNN-LSTM architecture and illustrates the sequential flow of data through its different layers. The process begins with the input layer, which receives the raw time series data and forwards it to a series of convolutional layers. These layers apply filters that extract key features by analyzing short time windows. Once the features have been extracted, they are further refined through a max pooling layer that reduces dimensionality while preserving essential information. The processed data is then passed to an LSTM layer, which learns sequential dependencies and captures long-term trends in the time series. In the final stage, the fully connected layer interprets the learned representations before producing the final prediction.

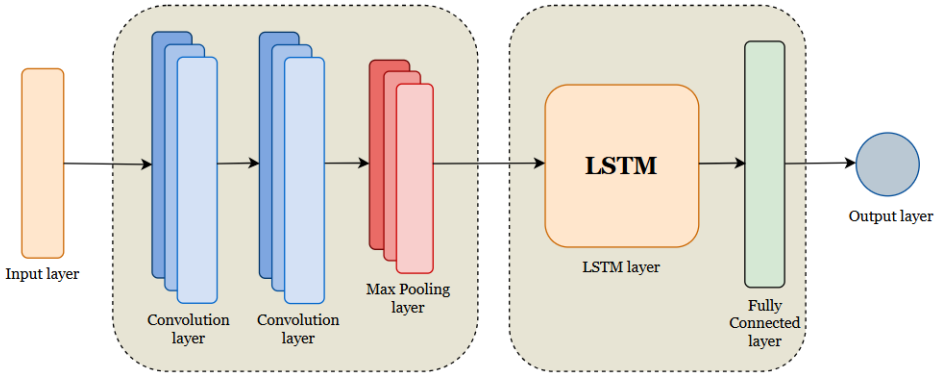
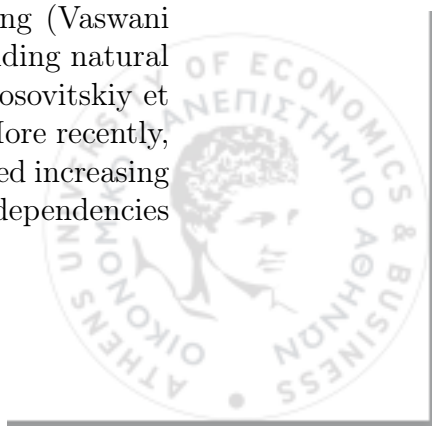


FIGURE 6.16: CNN-LSTM structure diagram.

6.3 Transformers

The development of transformer architectures in deep learning (Vaswani et al., 2017) has significantly advanced multiple domains including natural language processing (Devlin et al., 2018), computer vision (Dosovitskiy et al., 2020) and speech processing (Dong, Xu, and Xu, 2018). More recently, the potential of transformers in time series forecasting has gained increasing attention due to their ability to effectively capture long-range dependencies and model complex interactions in sequential data.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Unlike recurrent models, transformers do not process data sequentially. Instead, they leverage self-attention mechanisms to directly access any part of the input history, allowing them to model long-term dependencies more effectively. This characteristic makes them a promising alternative for time series forecasting, where capturing both short-term fluctuations and long-term trends is essential.

However, significant challenges remain when adapting transformers for time series forecasting. One critical challenge is the effective capture of both short-range fluctuations and long-term trends, alongside seasonality patterns (Wu et al., 2021). Additionally, transformers inherently suffer from quadratic computational complexity due to the self-attention mechanism, complicating their application to high-dimensional or very long sequences typically found in time series data. To address these challenges, recent research has introduced optimized architectures, such as sparse attention mechanisms and memory-efficient variants. While transformer-based models for time series forecasting remain an active research area, reviews have addressed related aspects like forecasting methods (Lim and Zohren, 2021) and classification techniques (Ismail Fawaz et al., 2019).

6.3.1 Self-Attention Mechanism

The core innovation in transformers is the self-attention mechanism, specifically scaled dot-product attention. Given query matrix $Q \in \mathbb{R}^{n \times d_k}$, key matrix $K \in \mathbb{R}^{n \times d_k}$ and value matrix $V \in \mathbb{R}^{n \times d_v}$, self-attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{6.25}$$

Here, the term $\frac{1}{\sqrt{d_k}}$ scales the dot product to stabilize gradients during training by preventing excessively large values.

6.3.2 Multi-Head Attention

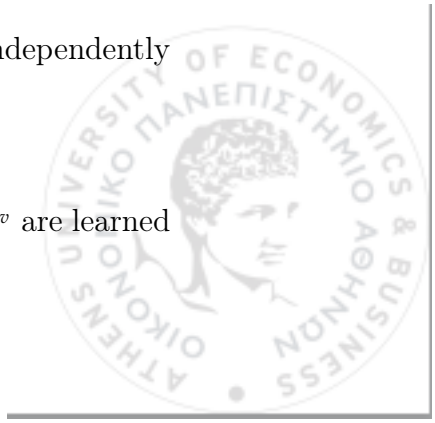
Multi-head attention allows transformers to simultaneously capture diverse information from different representation subspaces. It involves parallel attention computations, each with distinct learned linear projections from the original input X. Specifically, the multi-head attention is defined as:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{6.26}$$

Each head_i, referred to as an *attention head*, is computed independently as:

$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, and $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ are learned



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

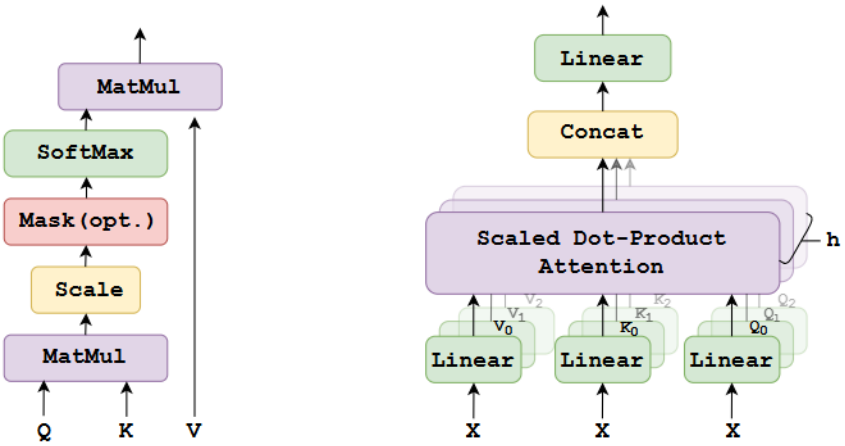


FIGURE 6.17: Scaled Dot-Product Attention (left) and Multi-Head Attention (right).

parameter matrices specific to each head, and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ is a final projection matrix combining outputs from all attention heads. Figure 6.17 illustrates the scaled dot-product attention along with the multi-head attention mechanism, which consists of several attention layers operating in parallel.

6.3.3 Positional Encoding

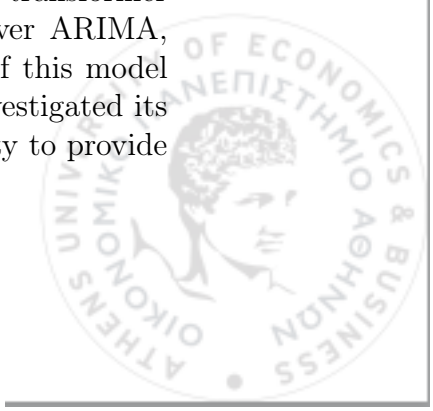
Because transformers do not inherently encode sequential order, explicit positional encodings must be introduced to inform the model about input positions. Vaswani et al. (2017) proposed sinusoidal positional encodings:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Here, pos denotes input positions, and i indexes the dimension. The periodic nature of these encodings allows transformers to generalize to sequence lengths beyond those observed during training.

Transformer for Univariate Time Series Forecasting

Li et al. (2019) showed that a full encoder-decoder transformer outperformed traditional statistical approaches, RNN-based methods and a matrix factorization method. Wu et al. (2020) also applied a transformer model to forecast influenza trends and saw improvements over ARIMA, LSTM and GRU-based sequence models. The architecture of this model is illustrated in Figure 6.18. Meanwhile, Lim et al. (2020) investigated its application in multi-horizon forecasting, emphasizing its ability to provide insights into temporal patterns.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

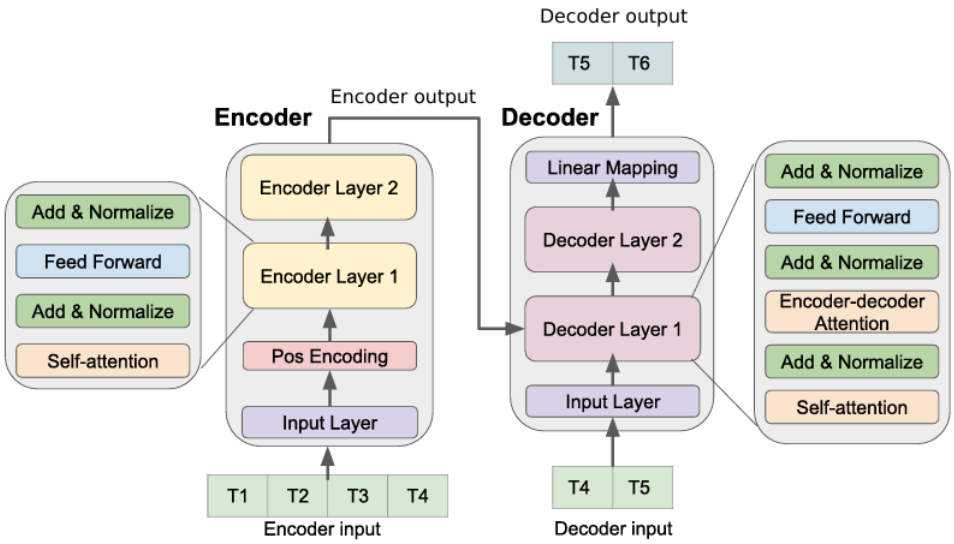


FIGURE 6.18: Transformer-based forecasting model architecture. Source: Wu et al. (2020). The model uses an encoder-decoder structure: the encoder processes the input sequence (e.g., past time steps T1–T4) using self-attention and feed-forward layers, generating contextual embeddings. These are passed to the decoder, which generates future values (e.g., T5–T6) using both self-attention and encoder-decoder attention. Positional encoding is added to retain sequence order information.

Despite these successes, transformers are not yet a universal solution for univariate time series forecasting but they offer considerable potential when properly adapted. Their performance depends on factors such as model architecture, dataset characteristics and the nature of the forecasting task. As researchers continue refining these models, transformer-based forecasting is likely to become an increasingly powerful alternative to traditional methods, especially for tasks requiring long-term trend analysis and high-dimensional data integration.


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Chapter 7

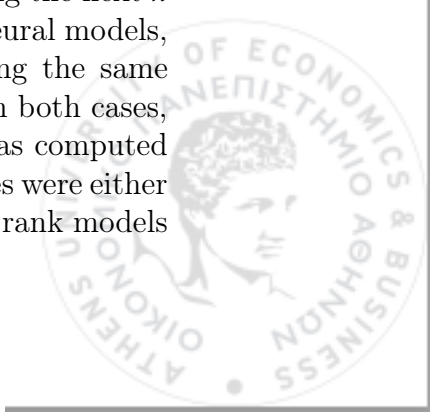
Practical Evaluation of the Forecasting Models

As already stated, time series forecasting plays a crucial role in many real-world domains including finance, meteorology and industry. The field has evolved significantly over the years with a wide array of models emerging, from classic statistical methods to modern deep learning approaches, each designed to improve predictive performance. But while model development has advanced rapidly, evaluating these models fairly remains a challenge. In many research settings, the datasets used for benchmarking are carefully selected, sometimes out of necessity, to reduce computational costs or simplify analysis. As a result, evaluations often rely on a small number of datasets, many of which fail to reflect the diversity and complexity of real-world data. In some cases, only subsets of larger datasets are used, limiting the scope of analysis. Furthermore, comparisons are frequently made between just a few models, with inconsistent levels of tuning and optimization across them, which further complicates the fairness of the evaluation.

These limitations can introduce bias, whether intentional or not. For example, when only four datasets are selected, studies have shown that nearly half of the models tested can be identified as the best-performing and over three-quarters may rank within the top three. This shows how easily model rankings can be influenced by dataset selection.

To address these concerns, this chapter takes a more careful and transparent approach to model evaluation. The findings mentioned earlier are taken into account and provide the basis for the analysis that follows. The aim is not to critique how evaluations are typically conducted, but to support a fair and meaningful comparison of the forecasting models discussed in the previous chapters.

The experiments were implemented in Python using two complementary workflows. The first focused on classical models, trained and evaluated with the *statsforecast* library using a single holdout evaluation, where each model was trained up to time $T - h$ and tasked with forecasting the next h steps, with h predefined per dataset. The second focused on neural models, developed with the *neuralforecast* library and evaluated using the same procedure, with training accelerated on an RTX-3080 GPU. In both cases, the Symmetric Mean Absolute Percentage Error (SMAPE) was computed independently for each time series. The resulting SMAPE values were either averaged across all series to assess overall accuracy, or used to rank models



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

within dataset–frequency pairs, with final comparisons based on the mean rank across groups. Data handling was performed using the `pandas` and `numpy` libraries.

7.1 Evaluation Strategy

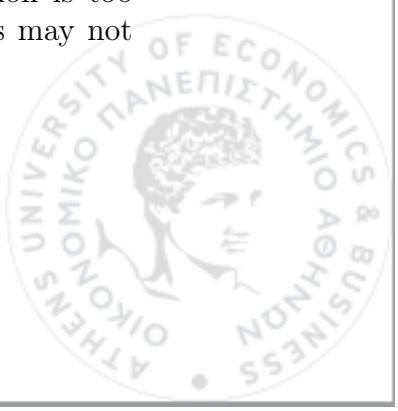
The evaluation is organized into four main steps: selecting the datasets, choosing the models to compare, measuring their performance and analyzing how they perform across different settings.

The first step is to bring together a collection of benchmark datasets. These datasets are chosen to reflect a wide range of characteristics. They differ in frequency, length and structure and include various patterns like trend, seasonality and irregularity. Next, a group of forecasting models is selected. These include both traditional statistical approaches, like *ARIMA* and *ETS*, and more recent deep learning models, such as *Informer*, *NHITS* and *TiDE*. The idea is to evaluate models with different foundations to better understand how each one performs under a shared set of conditions. The full list of models used in the experiment is provided in Table 7.1. The third step involves measuring how well each model performs on each dataset. For every combination of model and dataset, the Symmetric Mean Absolute Percentage Error (SMAPE) is calculated, which gives a standardized way to compare accuracy. Models are then ranked on each dataset based on their SMAPE values.

The final part of the evaluation takes a closer look at how each model performs across different datasets. The aim is to understand how performance varies in practice and to get a clearer picture of how models behave under different conditions. To start, each model is evaluated on the full set of datasets to establish a baseline ranking. This gives an overall sense of which models tend to perform better when all datasets are considered together. Then, for each model, the datasets where it performs best are identified. Finally, model rankings are tracked as more datasets are added to the evaluation. This helps determine whether a model's strong performance remains consistent across different types of data or if it tends to rely on specific cases to perform well.

7.1.1 Dataset Selection

The datasets used in time series forecasting can significantly affect how meaningful and trustworthy the results are. Because time series often vary in their structure, length and overall behavior, it is important to choose datasets that match the goals of the evaluation. If the selection is too limited or not representative of real-world data, the conclusions may not generalize well beyond the test setting.

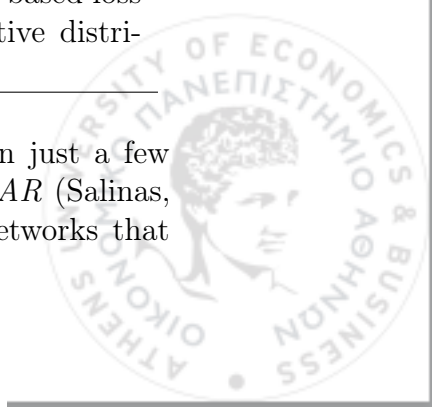


Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

TABLE 7.1: Nixtla forecasting models with full names

Acronym	Full Name	Description
ARIMA	AutoRegressive Integrated Moving Average (AutoARIMA, StatsForecast)	Automatically selects the best ARIMA(p,d,q) parameters by minimizing an information criterion such as AIC or BIC.
ETS	Exponential Smoothing State Space Model (AutoETS, StatsForecast)	Exponential smoothing model that automatically selects error, trend, and seasonality components.
SES	Simple Exponential Smoothing (StatsForecast)	Forecasts using a weighted average of past observations with exponentially decaying weights.
SNaive	Seasonal Naïve (StatsForecast)	Baseline method that repeats the last observed values from the same season.
TCN	Temporal Convolutional Network (NeuralForecast)	Deep learning model using dilated causal convolutions and residual connections to capture long-term dependencies.
RNN	Recurrent Neural Network (NeuralForecast)	Autoregressive forecasting model based on Elman RNN cells, processing sequences with hidden state recurrence.
NHITS	Neural Hierarchical Interpolation for Time Series (NeuralForecast)	Specializes its partial outputs in the different frequencies of the time series through hierarchical interpolation and multi-rate input processing.
Informer	Informer: Efficient Transformer for Long Sequence Forecasting (NeuralForecast)	Transformer-based model for long sequence forecasting.
TiDE	Time-series Dense Encoder (NeuralForecast)	An MLP-based univariate time-series forecasting model. TiDE uses MLPs in an encoder-decoder structure for long-term forecasting.
DeepAR	Deep AutoRegressive Model (NeuralForecast)	Probabilistic autoregressive forecasting model based on RNNs, trained with likelihood-based loss to produce full predictive distributions.

In recent work, forecasting models are often evaluated on just a few datasets, typically between three and six. For instance, *DeepAR* (Salinas, Flunkert, and Gasthaus, 2019), a model based on LSTM networks that



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

produces probabilistic forecasts, has been tested on datasets such as Parts, Electricity and Traffic. These datasets are widely used because they are publicly available and easy to compare across different methods.

The dataset choice usually depends on what the model is designed to do. Models like *Informer* (Zhou et al., 2021), *NHITS* (Challu et al., 2023), and *TiDE* (Das et al., 2024) are built for long-term forecasting. *Informer* is a transformer-based model designed to work with long input sequences more efficiently. *NHITS* builds on earlier ideas by using multiple processing steps at different time scales to improve long-range accuracy. Its design is based on MLPs, similar to its predecessor N-BEATS (Oreshkin et al., 2020), which uses stacked fully connected layers to predict future values directly from raw time series. *TiDE* also uses a structure based on MLPs, combining an encoder and a decoder to make forecasts, and is particularly focused on long-horizon tasks. An overview of the forecasting models used is shown in Table 7.1.

These models are usually tested on datasets that include long and continuous time series, such as Electricity, Traffic, Weather and the ETT series. For example, ETT contains a small number of time series with many time steps which makes it useful for studying models that need to capture long-term patterns.

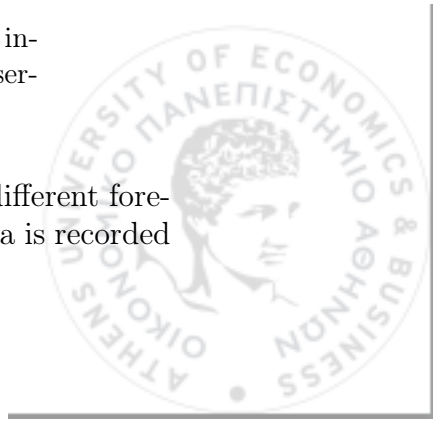
Evaluation setups have also changed over time. While *NHITS* is mainly tested on long-horizon forecasting tasks, *N-BEATS* was originally evaluated on more traditional benchmarks like Tourism, M3 and M4. These older datasets contain many time series but each one is shorter.

What's interesting about *TiDE* is that it does not focus on only one type of task. In addition to long-range forecasting, it also includes experiments on demand prediction using the M5 dataset. In this case, the model is compared with others such as *DeepAR*. This kind of evaluation provides a more complete view of how the model performs under different real-world conditions.

Dataset	Contents	Frequency	# Time Series	Total Observations	Horizon (h)	Seasonality
ETTh1	Electricity Load	Hourly	1	14400	48	24×365
ETTh2	Electricity Load	Hourly	1	14400	48	24×365
Labour	Employment stats	Monthly	57	28671	18	12
M3	Mixed domains	Monthly	1428	167562	18	12
		Quarterly	756	37004	8	4
		Yearly	645	18319	6	1
M4	Mixed domains	Monthly	48000	11246411	18	12
		Quarterly	24000	2406108	8	4
		Yearly	23000	858458	6	1
M5	Retail sales	Daily	30490	47649940	30	365
Traffic	Road occupancy	Daily	207	75762	30	365
Wiki2	Page views	Daily	199	72834	30	365
Total	–	–	128784	62589869	–	–

TABLE 7.2: Summary of datasets used in the analysis, including their frequency, number of time series, total observations, forecasting horizon, and seasonality.

In this analysis, a variety of datasets is used to represent different forecasting situations. The term frequency refers to how often data is recorded



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

for example, once a year, every month, every day or every hour. The datasets included are ETTh1, ETTh2, M3, M4, M5, Labour, Traffic and Wiki2. An overview of these datasets is shown in Table 7.2. For ETTh1 and ETTh2, only the training part of the data was used, which includes 14,400 hourly observations for each dataset. The forecasting horizon (h) varies across datasets, ranging from short-term (e.g., 6 steps for yearly data) to longer-term forecasts such as 48 hours ahead for the ETTh datasets or 30 days ahead for daily datasets like M5 and Traffic. Seasonality is also dataset-dependent and reflects recurring patterns based on the data frequency. For instance, yearly seasonality is captured as 1 for annual data, 12 for monthly data, and 365 for daily data.

7.1.2 Evaluation Metric

There are many ways to measure how well a forecasting model performs. Some metrics depend on the scale of the data, while others are based on percentage or relative error. Common choices include Mean Absolute Error (MAE), Root Mean Squared Error (RMSE) and SMAPE. Each one offers different aspects of model performance and the best choice depends on the situation. SMAPE is especially useful when working with series that differ in scale, which is why it was used in competitions like M4.

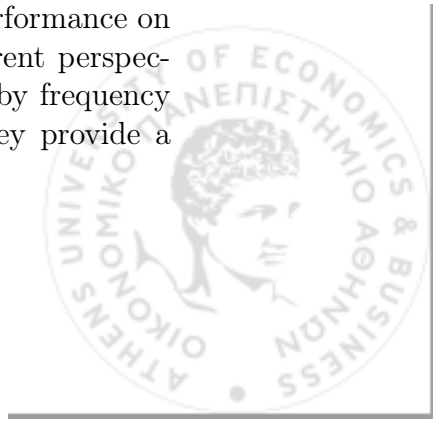
For this analysis, SMAPE is used as the main evaluation metric. It's defined as:

$$\text{SMAPE} = \frac{1}{n} \sum_{i=1}^{n} \frac{|\hat{y}_i - y_i|}{(|\hat{y}_i| + |y_i|)\,/2} \tag{7.1}$$

where $\hat{y}_i$ is the i-th predicted value, y_i is the actual value of that point and n is the total number of points being forecasted. In the experiment of this thesis, each time series is split into a training set and a held-out horizon. The model forecasts this horizon, so $n = h$, where the value of h is determined by the frequency of the time series (see Table 7.2).

7.2 Results

Before turning to the figures, it is useful to recall the scope of this evaluation. The results presented here summarise performance across a wide variety of datasets, each with its own frequency, forecast horizon and structural complexity. Some datasets are large-scale and high-frequency, while others are smaller and exhibit slow-moving seasonal or trend components. Because of this diversity, no single figure can fully capture performance on its own. Instead, each plot in this section focuses on a different perspective: overall accuracy, stability across datasets, performance by frequency and behaviour across forecast horizons. Taken together, they provide a balanced view of which models perform well on average.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Figure 7.1 compares forecasting methods across the diverse benchmark described in Table 7.2. The models will be compared through their average rankings which are determined per model as follows:

1. The test horizon of every time series is forecast.
2. A SMAPE error is computed for each individual series using the forecasts.
3. These errors are aggregated within each dataset-frequency group, producing one average SMAPE value per group.
4. Models are compared within each group based on this average SMAPE and ranks are assigned.
5. The overall performance of the model is obtained as the average of its ranks across all groups.

Lower ranks indicate better performance, with rank 1 representing the top position. Models at the top are therefore not excelling in isolated cases but consistently producing accurate forecasts across a wide variety of datasets.

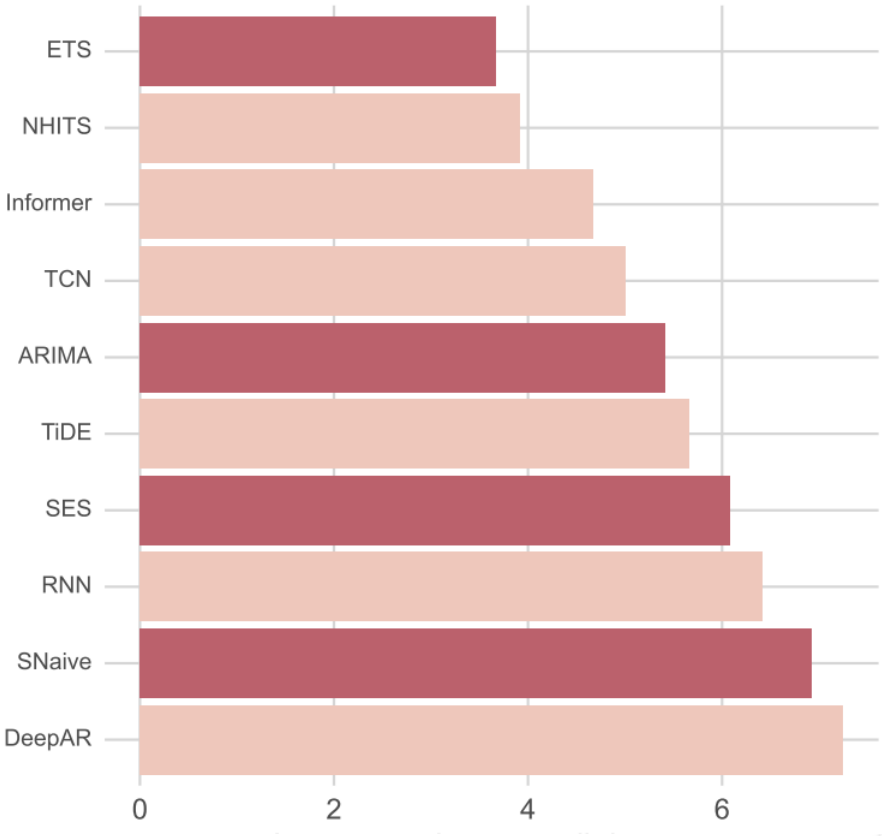


FIGURE 7.1: Average rank across all datasets. The darker red bars correspond to statistical methods, and the lighter peach bars correspond to neural network-based methods.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

ETS leads the ranking, showing that a well-tuned statistical model can still outperform more complex neural architectures. Its ability to capture both trend and seasonality without extensive parameterization gives it a strong advantage across diverse temporal patterns. Close behind, *NHITS* leverages its hierarchical, multi-resolution design to adapt to different time scales, performing strongly in both high-frequency and lower-frequency series. *Informer* follows, with its sparse attention mechanism enabling efficient use of long historical sequences, and *TCN* also performs well, particularly in data with clear and stable seasonal cycles.

In the middle tier, *ARIMA*, *TiDE*, *SES* and *RNN* deliver competitive results in certain contexts but lack the same level of consistency across the full benchmark. *SNaive* and *DeepAR* occupy the lower ranks. While *ARIMA* and *SNaive* are constrained by their simpler statistical assumptions, *DeepAR*'s weaker showing is likely due to a mismatch between its design and the heterogeneous nature of the datasets.

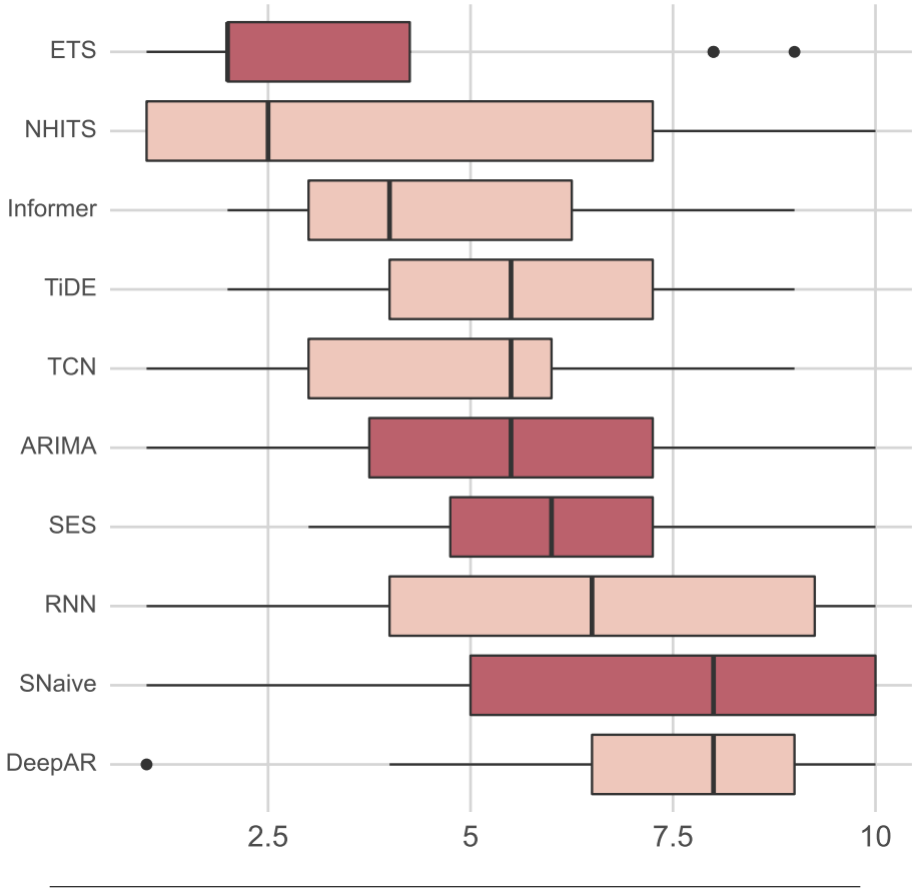


FIGURE 7.2: Rank distribution across all datasets.

Figure 7.2 presents, for each model, the distribution of the ranks from which the average rank was computed (see Step 4). Each box captures the interquartile range (IQR) of rankings, the central line marks the median and the whiskers extend to the broader range of performance. Lower medians

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

correspond to better typical performance, while narrower boxes indicate greater stability from one dataset to another.

ETS has both the lowest median rank and one of the tightest spreads, showing that its strong performance is not only high but also stable across a wide variety of forecasting tasks. Next, *NHITS* with a slightly higher median and a broader range, occasionally producing both top-tier and lower-end results, which points to higher sensitivity to dataset characteristics. *Informer*, *TiDE*, *TCN*, *ARIMA* and *SES* have moderate variability, with *Informer* achieving a lower median than the others in this group. *RNN* and *SNaive* show both higher median and wide spread, pointing to less competitive and less predictable outcomes. *DeepAR* ends up near the bottom of the rankings. Its high median score suggests that, for most datasets, it struggles to match the consistency and accuracy shown by the stronger performers.

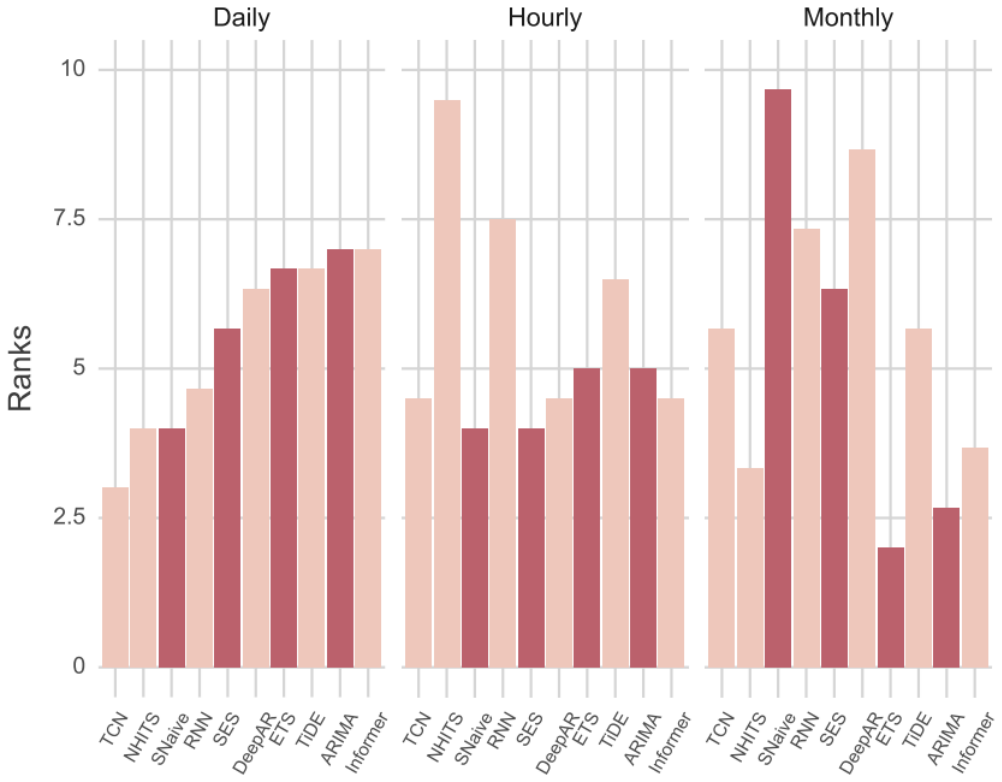


FIGURE 7.3: Performance comparison of forecasting models across different temporal frequencies (Part A).

.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

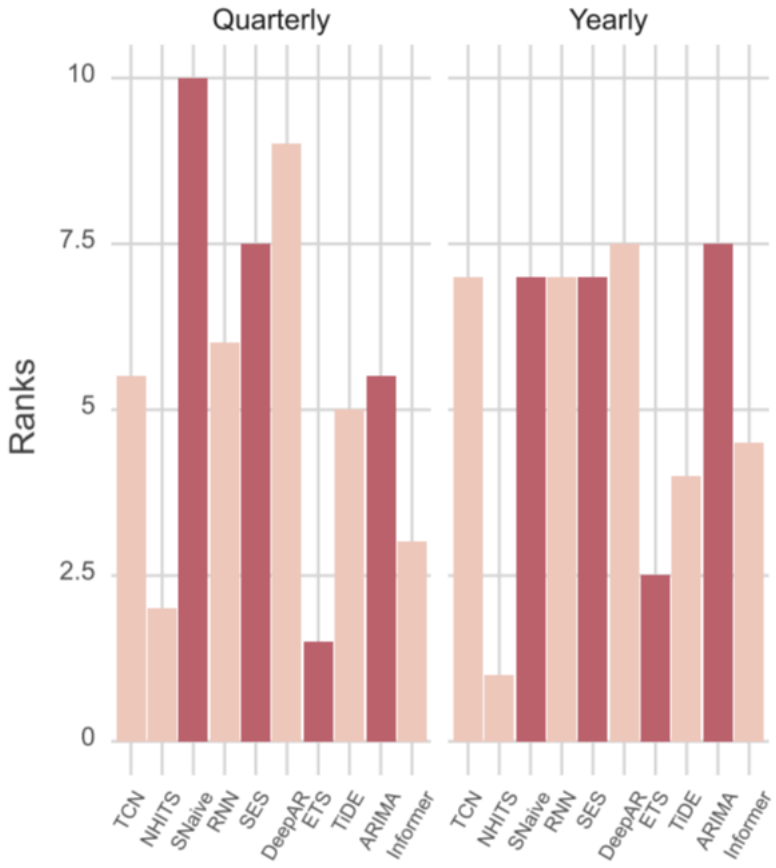


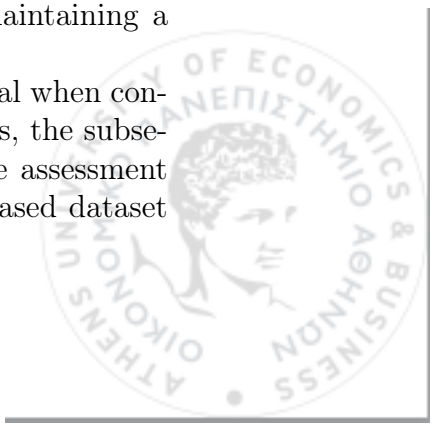
FIGURE 7.4: Performance comparison of forecasting models across different temporal frequencies (Part B).

.

Moving forward, Figures 7.3 - 7.4 compare forecasting models by averaging the ranks assigned in Step 4, but this time within each temporal frequency rather than across all datasets. The figures show results for five distinct frequencies: daily, hourly, monthly, quarterly, and yearly. Each bar represents the mean position a model achieved within that frequency.

TCN leads the daily frequency results, with *NHITS* and *SNaive* close behind, while methods such as *ARIMA* fall toward the lower end of the rankings. In the hourly frequency, *SES* and *SNaive* take the lead, and *ETS* moves into a noticeably stronger position. For monthly data, *ETS* clearly dominates, followed by *ARIMA* and *NHITS*, whereas *SNaive* has the weakest results. A similar pattern can be seen for quarterly data, with *ETS* again at the top and *NHITS* close behind. In the yearly frequency, *NHITS* delivers one of its strongest performances, with *ETS* still maintaining a competitive standing.

To support the decision to use eight datasets (twelve in total when considering their frequencies) in evaluating the forecasting models, the subsequent analysis focuses on how model rankings vary when the assessment is based on different subset sizes drawn from the frequency-based dataset collection, rather than from the full benchmark.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

For each model and each size $k \in \{1,\ldots,6\}$, the k frequency-based datasets where the model attains its highest relative rank compared to other models are selected. For brevity and to conserve space, the figures presented correspond to the *TCN*, *DeepAR*, and *Informer* models, and only their findings are discussed in detail (Figures 7.5, 7.6 and 7.7). For example, in the bottom-right bar plot of Figure 7.5, model *TCN* holds the highest average rank when all models are evaluated on the six frequency-based datasets where it achieves its best relative ranking. This shows that, through dataset selection, every model can consistently place among the top three, indicating that all models can be competitive depending on the specific datasets considered.

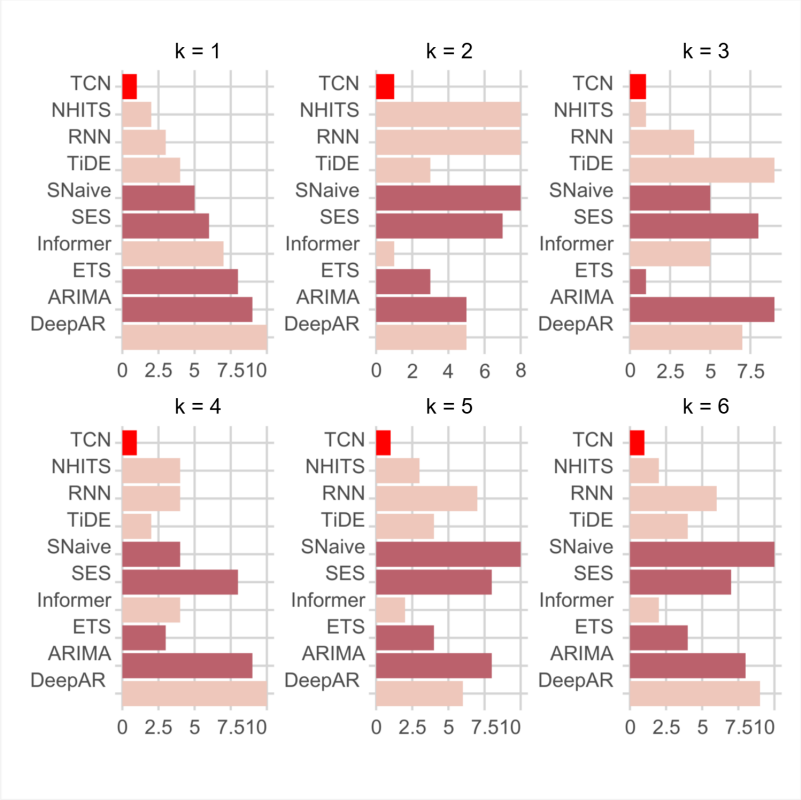


FIGURE 7.5: Variation of TCN's ranking across different dataset selections. For each $k = 1,\ldots,6$, we choose the n datasets where TCN achieves its highest relative performance. All models are trained on these selected datasets, and their average ranks are shown. The results demonstrate that depending on the chosen datasets, TCN can appear to be the best-performing model, even if this does not reflect its overall performance.


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

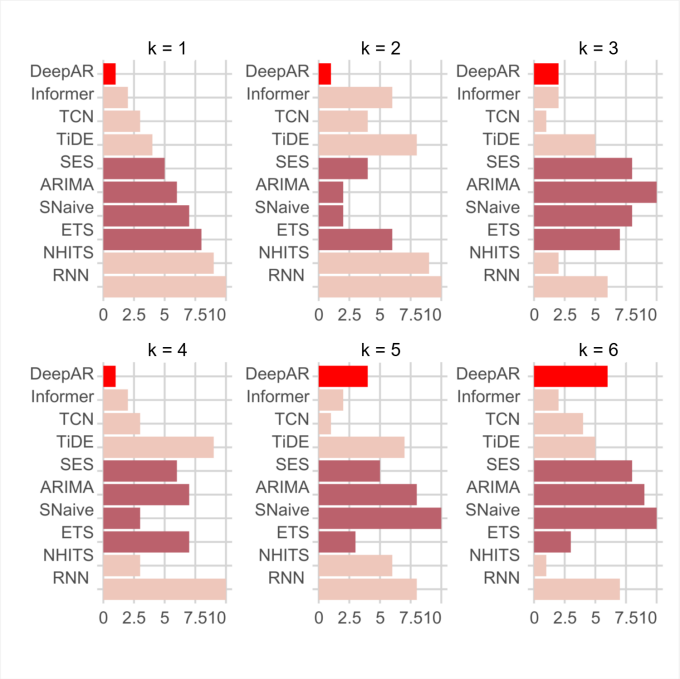


FIGURE 7.6: Variation of DeepAR's ranking across different dataset selections. Same analysis as in Figure 7.5, but for DeepAR.

.

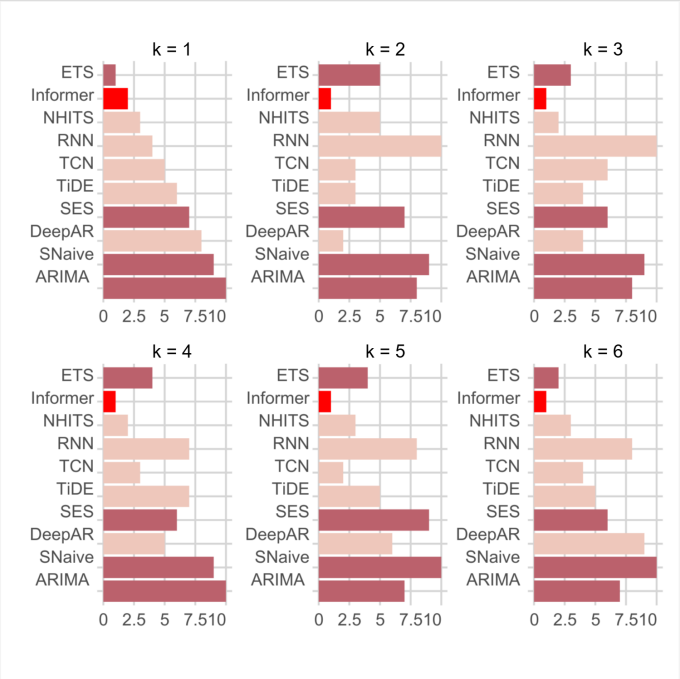


FIGURE 7.7: Variation of Informer's ranking across different dataset selections. Same analysis as in Figure 7.5, but for Informer.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Lastly, Figure 7.8 is further proof of how selective dataset choice can inflate the perceived performance of forecasting models. This outcome is a form of selection bias, in which the apparent strength of a model depends heavily on the specific datasets included in the evaluation rather than on its general forecasting ability. When only two datasets are considered, it becomes possible to position every model within the top two rankings for at least one dataset pair. Under such conditions, even models that perform modestly on average across the full benchmark can appear much more competitive simply by selecting favorable datasets. Increasing the number of datasets to three reduces this effect slightly, yet the pattern stays evident: 60% of the models still appear as the top-ranked option. This shows how easily rankings can be shaped by limited dataset coverage which raises concerns about the reliability of conclusions drawn from narrow benchmarks. Even with six datasets, the effect does not disappear. The results show that 70% of models still achieve a rank of second or higher when the evaluation is based on a carefully chosen subset. Although the magnitude of the bias decreases as the number of datasets grows, given the variability in model performance across different datasets, it can still produce inflated results.

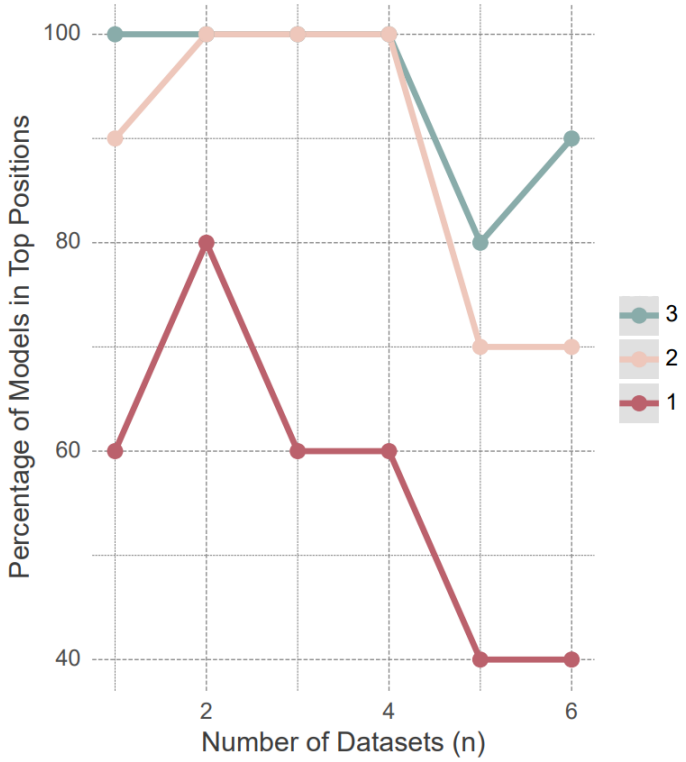
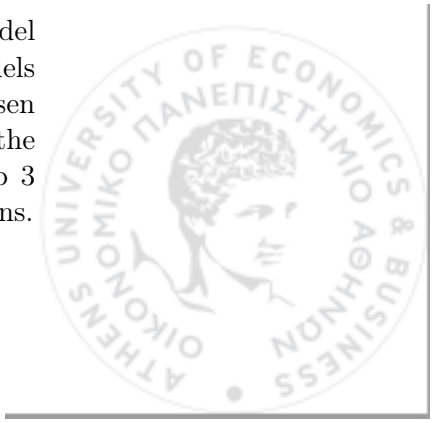


FIGURE 7.8: Impact of selective dataset choice on model ranking. The dark red line shows the percentage of models that achieve 1st place when evaluated on n datasets chosen in their favor. The light peach and green lines show the percentage of models that appear in the top 2 and top 3 positions, respectively, under similarly favorable selections.


ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

The patterns in Figure 7.8 show the risks of relying on small and hand-picked benchmarks. Strong results in such cases may not translate to broader forecasting scenarios and any conclusions drawn from them must be framed with clear recognition of these limitations. By expanding the scope of evaluation to include datasets with different domains, frequencies and structural complexities, model comparisons become more credible and less susceptible to the distortions introduced by selection bias.

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Chapter 8

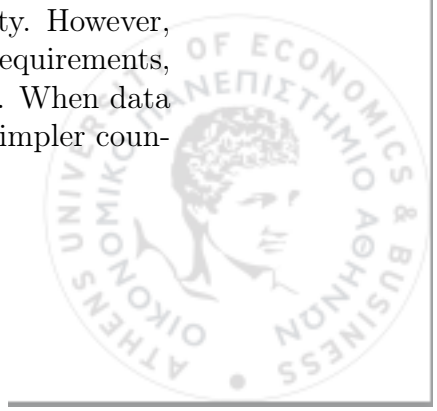
Conclusion and Recommendations

"All models are wrong, but some are useful."
- George Box -

This thesis set out to explore univariate time series forecasting from two angles: a broad review of forecasting methods and a side-by-side comparison of selected models. The central aim was not just to list algorithms but to understand how different approaches behave under varied forecasting conditions. The theoretical review covered a wide spectrum of methods from established statistical techniques to modern machine learning approaches, including decision trees, gradient-boosted ensembles such as XGBoost and advanced neural networks. While not all of these models were implemented in the experimental phase, their inclusion provided a broader conceptual framework and highlighted potential avenues for future methodological integration.

The experiments focused more narrowly on two main categories: classical statistical models and neural network–based models. These were chosen to represent two distinct philosophies. Classical models are interpretable, efficient and grounded in clear statistical principles. Neural networks, on the other hand, are flexible, data-driven systems capable of learning complex patterns. To ensure a fair comparison, all models were tested on the same datasets with the same preprocessing steps, cross-validation method and accuracy metrics. The datasets were purposely varied: some had clear seasonal patterns, others were more irregular, some were simple and others more complex.

The results made it clear that there is no single "best" model for all cases. Classical models often excel when the data has well-defined seasonality, stable trends and relatively low variability. In some cases, they match or even outperform neural networks particularly for shorter forecasts. Neural models are more capable when dealing with irregular seasonal structures, non-linear relationships or datasets with more variability. However, these advantages came at the cost of greater computational requirements, longer training times and the need for larger amounts of data. When data is limited, neural networks could struggle compared to their simpler counterparts.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

An important contribution of this work lies in its methodological consistency. By applying the same evaluation framework to both statistical and neural models, the comparisons avoided biases that might otherwise obscure genuine performance differences. This consistency is not only of academic interest but is also directly relevant to real-world forecasting practice. In operational environments where resources are constrained, selecting a model based on an inconsistent or overly optimistic evaluation can lead to significant inefficiencies or missed performance targets.

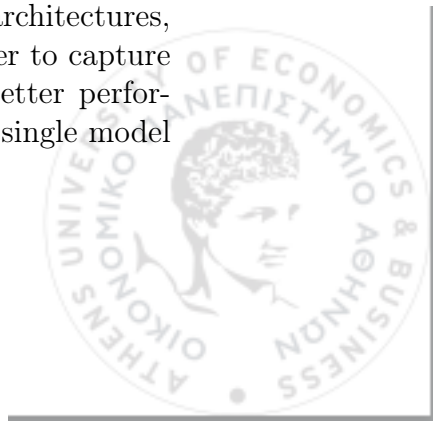
The findings align with the broader understanding in forecasting literature: model effectiveness is highly context-dependent, shaped by the interaction between data characteristics, forecast horizon, and available computational resources. The "best" choice depends on the specific forecasting task, the nature of the data, the forecast horizon, and the practical constraints under which the model will be deployed. Understanding these dependencies is crucial for making informed, context-aware model selections.

Recommendations for Future Work

Several directions present themselves for extending and enriching the work undertaken in this thesis. One immediate opportunity lies in expanding the experimental framework to include additional machine learning models and other non-neural approaches discussed in the theoretical chapters. These methods have demonstrated competitive performance in numerous predictive modelling contexts and could offer valuable insights into how intermediate-complexity models compare to both classical statistical techniques and deep learning architectures in time series forecasting.

A particularly promising direction is the extension of the study to multivariate time series forecasting, where multiple interrelated variables are predicted jointly. This setting introduces additional modelling challenges, including capturing cross-variable dependencies and shared temporal structures. While classical models can be adapted for multivariate forecasting, the increased complexity of the problem creates a setting where neural networks and advanced ensemble methods may have greater scope to demonstrate performance advantages. Conducting controlled experiments in this domain would not only test the scalability of the methods studied here but also help clarify the contexts in which neural approaches provide meaningful benefits over simpler alternatives.

Another area for exploration is the development and evaluation of hybrid forecasting frameworks that combine the strengths of different model classes. For example, a hybrid system might pair the interpretability and stability of statistical models with the flexibility of neural architectures, using one to model predictable seasonal patterns and the other to capture residual, non-linear components. Such designs could offer better performance across a wider range of forecasting scenarios than any single model type.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Beyond methodological extensions, future work could benefit from applying the evaluation framework developed in this thesis to specific application domains such as energy demand forecasting, financial market prediction, supply chain optimisation or healthcare monitoring. These fields often bring unique challenges, like sudden structural changes or unusual seasonal patterns, which could influence which models perform best. Testing across such domains could produce practical recommendations tailored to different industries.

Finally, future research could place greater emphasis on probabilistic forecasting, as examining prediction intervals alongside accuracy metrics would provide a more complete picture of model performance. In many decision-making contexts, understanding the range of possible outcomes can be just as important as identifying the most likely value, making this an essential complement to traditional point forecasts.

In summary, expanding the range of models, moving into multivariate forecasting, using hybrid and probabilistic approaches and applying the methodology to domain-specific challenges all represent valuable next steps. Pursuing these directions would deepen the empirical foundation for comparing forecasting methods.

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Bibliography

Alhussein, Musaed, Khursheed Khursheed, and Syed Haider (Jan. 2020). "Hybrid CNN-LSTM Model for Short-Term Individual Household Load Forecasting". In: *IEEE Access* 8, pp. 180544–180557. DOI: 10.1109/ACCESS.2020.3028281.

Altun, Murat et al. (Feb. 2023). "Monkeypox Detection Using CNN with Transfer Learning". In: *Sensors* 23, p. 1783. DOI: 10.3390/s23041783.

Bai, Shaojie, J Zico Kolter, and Vladlen Koltun (2018). "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling". In: *arXiv preprint arXiv:1803.01271*. URL: https://arxiv.org/abs/1803.01271.

Benidis, Konstantinos et al. (2020). "Neural forecasting: Introduction and literature overview". In: *CoRR* abs/2004.10240. arXiv: 2004.10240. URL: https://arxiv.org/abs/2004.10240.

Bezdan, Timea and Nebojsa Bacanin (Jan. 2019). "Convolutional Neural Network Layers and Architectures". In: pp. 445–451. DOI: 10.15308/Sinteza-2019-445-451.

Box, George E. P. and Gwilym M. Jenkins (1976). *Time Series Analysis: Forecasting and Control*. San Francisco: Holden-Day.

Box, George E. P. and Jenkins, Gwilym M. (1976). *AirPassengers dataset: Monthly totals of international airline passengers, 1949–1960*. https://www.rdocumentation.org/packages/datasets/versions/3.6.2/topics/AirPassengers. Available in R datasets package.

Breiman, Leo (1996). "Bagging predictors". In: *Machine Learning* 24.2, pp. 123–140. DOI: 10.1007/BF00058655. URL: https://doi.org/10.1007/BF00058655.

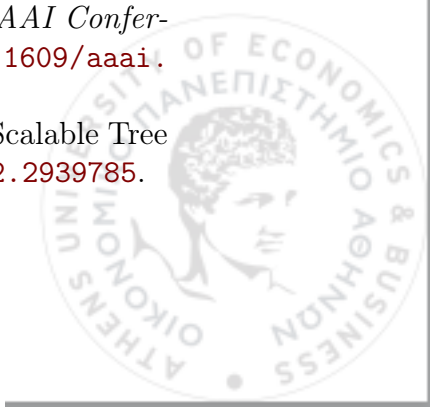
— (Oct. 2001). "Random Forests". In: *Machine Learning* 45, pp. 5–32. DOI: 10.1023/A:1010950718922.

Calin, Ovidiu (2020). *Deep Learning Architectures: A Mathematical Approach*. 1st. Springer Publishing Company, Incorporated. ISBN: 3030367207. URL: https://link.springer.com/book/10.1007/978-3-030-36721-3.

Chakraborty, Tanujit (n.d.). *Time Series Analysis*. URL: https://www.ctanujit.org/uploads/2/5/3/9/25393293/21_time_series_1.pdf.

Challu, Cristian et al. (June 2023). "NHITS: Neural Hierarchical Interpolation for Time Series Forecasting". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37, pp. 6989–6997. DOI: 10.1609/aaai.v37i6.25854.

Chen, Tianqi and Carlos Guestrin (Aug. 2016). "XGBoost: A Scalable Tree Boosting System". In: pp. 785–794. DOI: 10.1145/2939672.2939785.



Attribution-NonCommercial-ShareAlike 4.0 International https://doi.org/10.26219/heal.aueb.9410
http://creativecommons.org/licenses/by-nc-sa/4.0/

Cho, Kyunghyun et al. (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://doi.org/10.3115/v1/D14-1179.

Crone, Sven F., Michèle Hibon, and Konstantinos Nikolopoulos (2011). "Advances in forecasting with neural networks? Empirical evidence from the NN3 competition on time series prediction". In: *International Journal of Forecasting* 27.3, pp. 635–660. ISSN: 0169-2070. URL: https://www.sciencedirect.com/science/article/pii/S0169207011000616.

Das, Abhimanyu et al. (2024). *Long-term Forecasting with TiDE: Time-series Dense Encoder*. arXiv: 2304.08424 [stat.ML]. URL: https://arxiv.org/abs/2304.08424.

Devlin, Jacob et al. (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805. arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

Dong, Linhao, Shuang Xu, and Bo Xu (Apr. 2018). "Speech-Transformer: A No-Recurrence Sequence-to-Sequence Model for Speech Recognition". In: pp. 5884–5888. DOI: 10.1109/ICASSP.2018.8462506.

Dosovitskiy, Alexey et al. (2020). "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *CoRR* abs/2010.11929. arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

Dudek, Grzegorz (Jan. 2015). "Short-Term Load Forecasting Using Random Forests". In: vol. 323, pp. 821–828. ISBN: 978-3-319-11309-8. DOI: 10.1007/978-3-319-11310-4_71.

Elman, Jeffrey L. (1990). "Finding Structure in Time". In: *Cognitive Science* 14.2, pp. 179–211. DOI: 10.1207/s15516709cog1402_1.

Fernandez, Raul et al. (Sept. 2014). "Prosody Contour Prediction with Long Short-Term Memory, Bi-Directional, Deep Recurrent Neural Networks". In: DOI: 10.21437/Interspeech.2014-445.

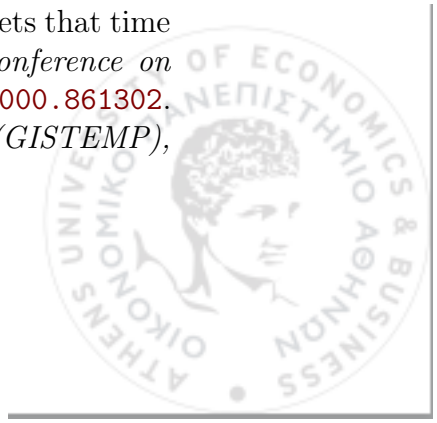
Fischer, Aurélie et al. (Sept. 2017). "Statistical learning for wind power: A modeling and stability study towards forecasting". In: *Wind Energy* 20. DOI: 10.1002/we.2139.

Friedman, Jerome H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine". In: *The Annals of Statistics* 29.5, pp. 1189–1232. URL: http://www.jstor.org/stable/2699986.

Garbin, Christian, Xingquan Zhu, and Oge Marques (May 2020). "Dropout vs. batch normalization: an empirical study of their impact to deep learning". In: *Multimedia Tools and Applications* 79, pp. 1–39. DOI: 10.1007/s11042-019-08453-9.

Gers, Felix and Jurgen Schmidhuber (Feb. 2000). "Recurrent nets that time and count". In: *Proceedings of the International Joint Conference on Neural Networks* 3, 189 –194 vol.3. DOI: 10.1109/IJCNN.2000.861302.

GISTEMP Team (2024). *GISS Surface Temperature Analysis (GISTEMP), Version 4*. https://data.giss.nasa.gov/gistemp/.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press. URL: http://www.deeplearningbook.org.

Guerdan, Luke (2019). *Diving Into Temporal Convolutional Networks*. URL: https://lukeguerdan.com/blog/2019/intro-to-tcns/.

He, Kaiming et al. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. arXiv: 1502.01852 [cs.CV]. URL: https://arxiv.org/abs/1502.01852.

Hewamalage, Hansika, Christoph Bergmeir, and Kasun Bandara (2021). "Recurrent Neural Networks for Time Series Forecasting: Current status and future directions". In: *International Journal of Forecasting* 37.1, pp. 388–427. ISSN: 0169-2070. URL: https://www.sciencedirect.com/science/article/pii/S0169207020300996.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Hyndman, Rob J. et al. (2023). *forecast: Forecasting functions for time series and linear models*. R package version 8.21. URL: https://CRAN.R-project.org/package=forecast.

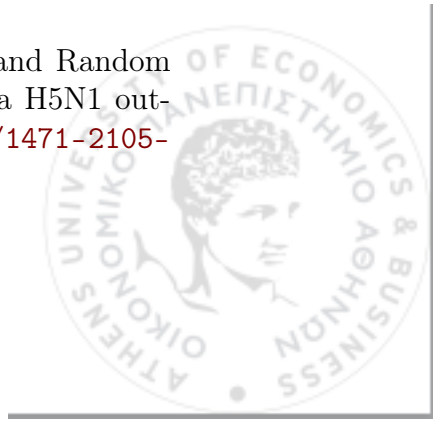
Hyndman, Rob J. and Athanasopoulos, George (2023). *qcement: Quarterly Australian Portland Cement Production (1956 Q1–2014 Q1)*. R dataset in the fpp2 package. Source: Australian Bureau of Statistics Cat. 8301.0.55.001. URL: https://www.rdocumentation.org/packages/fpp2/versions/2.5/topics/qcement.

Ioffe, Sergey and Christian Szegedy (Feb. 2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In.

Ismail Fawaz, Hassan et al. (July 2019). "Deep learning for time series classification: a review". In: *Data Mining and Knowledge Discovery* 33. DOI: 10.1007/s10618-019-00619-1.

Iwana, Bryan K. and Seiichi Uchida (2021). "An Empirical Survey of Data Augmentation for Time Series Classification with Neural Networks". In: *PLOS ONE* 16.7, e0254841. DOI: 10.1371/journal.pone.0254841. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0254841.

Kane, Michael et al. (Aug. 2014). "Comparison of ARIMA and Random Forest time series models for prediction of avian influenza H5N1 outbreaks". In: *BMC bioinformatics* 15, p. 276. DOI: 10.1186/1471-2105-15-276.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Laarhoven, Twan (June 2017). "L2 Regularization versus Batch and Weight Normalization". In: DOI: 10.48550/arXiv.1706.05350.

Lahouar, Ali and J. Ben Hadj Slama (Mar. 2015). "Random forests model for one day ahead load forecasting". In: DOI: 10.1109/IREC.2015.7110975.

LeCun, Yann et al. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. DOI: 10.1109/5.726791. URL: https://doi.org/10.1109/5.726791.

Lee, Shin-Jye et al. (Jan. 2018). "Image Classification Based on the Boost Convolutional Neural Network". In: *IEEE Access* PP, pp. 1–1. DOI: 10.1109/ACCESS.2018.2796722.

Lenail, Alex (2019). *NN-SVG: Publication-ready neural network architecture schematics*. URL: https://alexlenail.me/NN-SVG/index.html.

Li, Shiyang et al. (June 2019). *Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting*. DOI: 10.48550/arXiv.1907.00235.

Lim, Bryan and Stefan Zohren (Feb. 2021). "Time-series forecasting with deep learning: a survey". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379.2194, p. 20200209. ISSN: 1471-2962. DOI: 10.1098/rsta.2020.0209. URL: http://dx.doi.org/10.1098/rsta.2020.0209.

Lim, Bryan et al. (2020). *Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting*. arXiv: 1912.09363 [stat.ML]. URL: https://arxiv.org/abs/1912.09363.

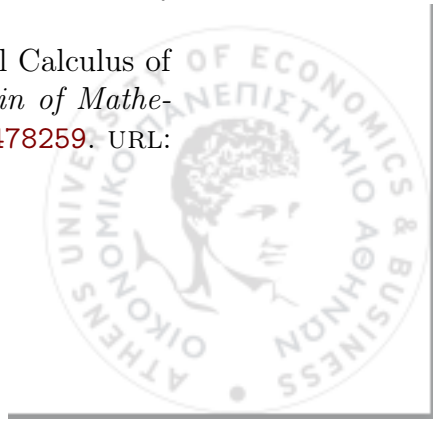
Livieris, Ioannis, Emmanuel Pintelas, and P. Pintelas (Dec. 2020). "A CNN-LSTM model for gold price time series forecasting". In: *Neural Computing and Applications* 32. DOI: 10.1007/s00521-020-04867-x.

Makridakis, Spyros, Evangelos Spiliotis, and Vassilios Assimakopoulos (Mar. 2018). "Statistical and Machine Learning forecasting methods: Concerns and ways forward". In: *PLOS ONE* 13.3, pp. 1–26. URL: https://doi.org/10.1371/journal.pone.0194889.

— (2022). "M5 accuracy competition: Results, findings, and conclusions". In: *International Journal of Forecasting* 38.4. Special Issue: M5 competition, pp. 1346–1364. ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2021.11.013. URL: https://www.sciencedirect.com/science/article/pii/S0169207021001874.

Masini, Ricardo, Marcelo Medeiros, and Eduardo Mendes (July 2021). "Machine Learning Advances for Time Series Forecasting". In: *Journal of Economic Surveys* 37. DOI: 10.1111/joes.12429.

McCulloch, Warren S. and Walter H. Pitts (1943). "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The Bulletin of Mathematical Biophysics* 5.4, pp. 115–133. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.



Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Moon, Jihoon, Yongsung Kim, and Eenjun Hwang (Nov. 2018). "Hybrid Short-Term Load Forecasting Scheme Using Random Forest and Multilayer Perceptron". In: *Energies* 11, p. 3283. DOI: 10.3390/en11123283.

Oreshkin, Boris N. et al. (2020). *N-BEATS: Neural basis expansion analysis for interpretable time series forecasting*. arXiv: 1905.10437 [cs.LG]. URL: https://arxiv.org/abs/1905.10437.

Petropoulos, Fotios, Rob J. Hyndman, and Christoph Bergmeir (2018). "Exploring the sources of uncertainty: Why does bagging for time series forecasting work?" In: *European Journal of Operational Research* 268.2, pp. 545–554. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2018.01.045. URL: https://www.sciencedirect.com/science/article/pii/S037722171830081X.

Pilario, Karl Ezra, Yi Cao, and Mahmoud Shafiee (May 2020). "A Kernel Design Approach to Improve Kernel Subspace Identification". In: *IEEE Transactions on Industrial Electronics* PP, pp. 1–1. DOI: 10.1109/TIE.2020.2996142.

Rahman, Md. Mijanur et al. (Feb. 2021). "Prospective Methodologies in Hybrid Renewable Energy Systems for Energy Prediction Using Artificial Neural Networks". In: *Sustainability* 13, p. 2393. DOI: 10.3390/su13042393.

Roque, Luis et al. (Dec. 2024). *Cherry-Picking in Time Series Forecasting: How to Select Datasets to Make Your Model Shine*. DOI: 10.48550/arXiv.2412.14435.

Rosenblatt, Frank (1958). "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65.6, pp. 386–408. DOI: 10.1037/h0042519. URL: https://doi.org/10.1037/h0042519.

Salinas, David, Valentin Flunkert, and Jan Gasthaus (2019). *DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks*. arXiv: 1704.04110 [cs.AI]. URL: https://arxiv.org/abs/1704.04110.

Spiliotis, Evangelos (2022). "Decision Trees for Time-Series Forecasting". In: *Foresight: The International Journal of Applied Forecasting* 64, pp. 30–44.

Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

Talavera, Edgar et al. (June 2022). *Data Augmentation techniques in time series domain: A survey and taxonomy*. DOI: 10.48550/arXiv.2206.13508.

Vaswani, Ashish et al. (2017). "Attention Is All You Need". In: *CoRR* abs/1706.03762. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

Wen, Wenjie et al. (2021). "Time Series Data Augmentation for Deep Learning: A Survey". In: *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 4627–4633. URL: https://www.ijcai.org/proceedings/2021/0631.pdf.

Attribution-NonCommercial-ShareAlike 4.0 International
http://creativecommons.org/licenses/by-nc-sa/4.0/
https://doi.org/10.26219/heal.aueb.9410

Wu, Haixu et al. (June 2021). *Autoformer: Decomposition Transformers with Auto-Correlation for Long-Term Series Forecasting*. DOI: 10.48550/arXiv.2106.13008.

Wu, Neo et al. (Jan. 2020). *Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case*. DOI: 10.48550/arXiv.2001.08317.

Zhou, Haoyi et al. (2021). *Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting*. arXiv: 2012.07436 [cs.LG]. URL: https://arxiv.org/abs/2012.07436.