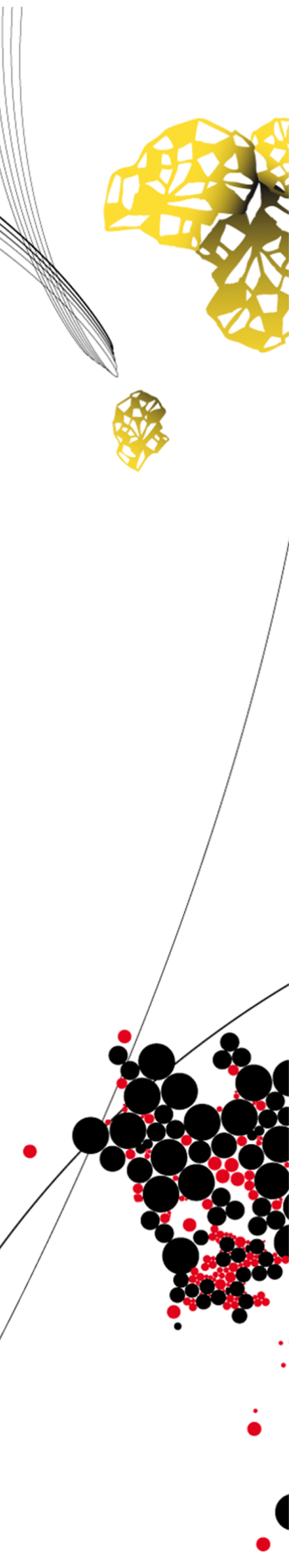

UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering, Mathematics & Computer Science**

# Master's Thesis

## A Comparison of Deep Neural Networks for Sales Forecasting

**Silvi Fitria**
**Master Applied Mathematics – Data Science**
**May 2023**

**UT Supervisor:**
Prof. Dr. Johannes Schmidt-Hieber
**Company Supervisor:**
Ali Akhani

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
Hallenweg 17
7500 NH Enschede
The Netherlands

UNIVERSITY OF TWENTE
FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS, AND COMPUTER SCIENCE
STATISTICS RESEARCH GROUP

---

Master's Thesis

# A Comparison of Deep Neural Networks for Sales Forecasting

---

Author
**Silvi Fitria**
EEMCS, Mathematics of Data Science

Chair and Supervisor
**Prof. Dr. Johannes Schmidt-Hieber**
EEMCS, Statistics

Committee Member
**Marcello Carioni**
EEMCS, Mathematics of Imaging & AI

Company Supervisor
**Ali Akhani**
Henkel

Date
**May 31, 2023**

# Preface

This thesis culminates my master's degree in Applied Mathematics at the University of Twente, which I started in September 2021. I have been working on this thesis for the past seven months, beginning in November 2022, and have overcome various challenges. This thesis was completed successfully with the generous support of many people.

The warm welcome at Henkel to guide me during the final phase of my master's studies marks the start of the internship and thesis. Their guidance and supervision supported me during the challenges of this time while also making it enjoyable, fascinating, and extremely beneficial. I would like to thank all my colleagues who are always open to discuss and to share their ideas in order to better understand the business process. I would especially want to thank Ali Akhani for allowing me the opportunity to complete my master's thesis at Henkel, as well as for his valuable support and guidance. I truly appreciate his daily supervision, who has spent a lot of time and effort providing me insightful feedback on the method, coding, and report.

Moreover, I would like to extend my gratitude to Johannes Schmidt-Hieber, my supervisor from the University of Twente for his constructive feedback, guidance, and flexibility. Also, I would like to thank Jan Schut as a Master's coordinator of Applied Mathematics, for his support in expediting the paperwork process and providing clear instructions for finishing this study. Furthermore, I would like to express my appreciation to my graduation committee, Marcello Carioni, for taking the time to review and to evaluate my thesis.

I would like to acknowledge the unwavering support of my mother, who has always been my pillar of strength and encouragement. Her endless motivation, support and blessings contributed to who I am today. In addition, many thanks to my aunt and uncle for their unconditional love and support throughout my journey. Finally, to my friends who have been my constant source of joy and occasional distraction, especially Vriendenkring who consists of Yesaya, Irine, Michael, Aldi, Raka, and Afif. I enjoy the many fun times we have shared together, and special thanks to them for proof-reading my thesis.

*Silvi Fitria*

# Abstract

Accurate sales forecasting plays an integral role in Supply Chain Management (SCM) for optimizing production and allocating resources, which leads to improved profitability. Since SCM comprises numerous products in a large number of stores, this research focuses on developing a global model that can be trained in a one-step process on all time series data. The application of this forecasting uses four novel Deep Neural Network (DNN) algorithms; namely Long Short-Term Memory (LSTM), Neural Basis Expansion Analysis for Time Series (NBEATS), Temporal Convolutional Network (TCN), and Transformer. It involves a comparative analysis based on their architectures, using real-world sales data in Henkel and simulated data. The TCN model performs best in both data, exhibiting the smallest evaluation error by taking almost two hours to train the global model. Implementing the best model resulted in an average improvement of 10% in forecast accuracy compared to the current forecasting method implemented in Henkel.

*Keywords:* Deep Neural Network, LSTM, NBEATS, Sales Forecasting, TCN, Time Series, Transformer.

# List of Abbreviations

| | |
|---|---|
| **CNN** | Convolutional Neural Network |
| **DL** | Deep Learning |
| **DNN** | Deep Neural Network |
| **FMCG** | Fast-Moving Consumer Goods |
| **GTIN** | Global Trade Item Number |
| **i.i.d** | identically and independently distributed |
| **IDH** | International Data Harmonization |
| **LSTM** | Long Short-Term Memory |
| **MAD** | Mean Absolute Deviation |
| **MAE** | Mean Absolute Error |
| **MBE** | Mean Bias Error |
| **ML** | Machine Learning |
| **MSE** | Mean Squared Error |
| **NBEATS** | Neural Basis Expansion Analysis for Time Series |
| **PDF** | Probability Density Function |
| **PMF** | Probability Mass Function |
| **ReLU** | Rectified Linear Unit |
| **RMSE** | Root Mean Squared Error |
| **RNN** | Recurrent Neural Network |
| **SCM** | Supply Chain Management |
| **SGD** | Stochastic Gradient Descent |
| **SMAPE** | Symmetric Mean Absolute Percentage Error |
| **TCN** | Temporal Convolutional Network |
| **WAPE** | Weighted Average Percentage Error |

# Contents

# Chapter 1

# Introduction

## 1.1 Background

Sales forecasting is an integral part of Supply Chain Management (SCM) since it has a substantial impact on corporate financial success. Accurate sales forecasting enables businesses to make well-informed manufacturing and logistical decisions, avoiding stock-outs or overstocking. (Loureiro et al., 2018). Businesses may benefit from accurately forecasting future sales by optimizing inventory levels, reducing waste, and minimizing inventory costs. Furthermore, effective sales forecasting allows companies to properly allocate resources and improve production schedules, ultimately leading to higher profitability, which positively benefits investors and increases company value (Agrawal & Schorling, 1996; Baecke et al., 2017). On the other hand, I.-F. Chen & Lu (2017) mentioned that poor sales forecasting might result in insufficient or overstocked inventories, causing the company to fail to meet customer needs and potentially harming profitability. Therefore, accurate sales forecasts are critical in bridging the gap between supply and demand as well as effectively planning its sales and operations (Gahirwal, 2013; Efat et al., 2022).

Sales forecasting involves utilizing historical sales data, product characteristics, and other relevant factors to predict short-term or long-term future sales performance of a business (Ma et al., 2016). In practice, sales forecasting entails acquiring and transforming raw data into structured information that can be analyzed to predict future sales, which relies on purchasing behaviour, promotional activities, and market conditions to develop the predictions (Paria et al., 2021; Jiménez et al., 2017). The major challenge in developing big data sales forecasting models is accurately modelling sparse and skewed data at the store and item levels (Ma & Fildes, 2021), especially given the large volume, variety, velocity, and veracity that pose several challenging problems for sales forecasting methods. More specifically, Q. Zhang et al. (2018) stated when learning how to employ large-scale models, locally optimum solutions may have a detrimental impact on traditional statistical and Machine Learning (ML) methods which cannot guarantee the convergence; they also require a high performance computing capability.

Traditional sales forecasting models are insufficiently detailed and adaptable to account for dynamic changes and non-linearities in sales time series at the store and product levels. Several techniques have been suggested to increase the accuracy of retail product sales forecasting. In recent years, most research has aimed to establish a universal or global forecasting strategy that can be applied to all sales time series under evaluation (Fildes et al., 2022). Deep Neural Network (DNN) method are proposed to model sales forecasting by extracting the complexity of spatiotemporal features from the data due to their ability to capture non-linear and complex correlations between variables, enabling the forecasting model to be robust to noise and time series length (Efat et al., 2022). DNNs have become increasingly used as adaptive parametric models capable of fitting complex data patterns (X.-W. Chen & Lin, 2014). Nevertheless, other types of DNNs are available, and it is still being determined which type is suitable for sales forecasting (Ma & Fildes, 2021). Therefore, the purpose of this research is to evaluate the performance of several DNN algorithms for sales forecasting and determine the most effective approach. The research findings will be helpful for businesses and academics interested in deploying DNNs for sales forecasting.

Applying novel Deep Learning (DL) approaches for sales forecasting brings several benefits to SCM. To begin with, it can improve the accuracy of sales forecasts when compared to standard statistical models. This might overcome the limitations of statistical forecasting approaches that rely heavily on linear relationships to predict future outcomes (Cecaj et al., 2020). DL techniques are able to model complicated non-linear relationships and learn to map inputs to outputs arbitrarily (Yang, 2021). Secondly, DL models can handle large amounts of complex data, including unstructured data, which provides valuable insights for sales forecasting and can be easily scaled to handle large volumes of data, making them suitable for large supply chain networks with many products and stores (Gamboa, 2017). The model has a rapid calculating speed and an excellent non-linear fitting capability (X. Zhang et al., 2018). Finally, DL models may automate forecasting, saving supply chain managers time and resources and allowing them to focus on other SCM issues such as logistics, procurement, and production planning (Jiménez et al., 2017).

In this study, the time series are arranged in a multi-level hierarchy; items are grouped into subcategories in a product taxonomy and are grouped into each store. In the context of retail or supply chain forecasting for product sales time series, a hierarchical structure can often be visualized as a tree, where the leaf nodes represent the finest granularity and the relationships between them are shown by the edges, reflecting the parent-child connections especially in Fast-Moving Consumer Goods (FMCG) supply chain (Paria et al., 2021). FMCG refers to a wide range of consumer goods that are typically sold quickly at a relatively low cost. These goods are often considered essential or daily necessities and are typically purchased frequently. Since Henkel is an FMCG company that produces various types of Laundry and Home Care products in numerous countries and also be sold in every store around the world, utilizing DNN techniques for sales forecasting would be a suitable method to manage such a complex and vast data landscape.

A comparison analysis is performed using novel DL approaches to demonstrate the effectiveness of the proposed model. This analysis is conducted on real-world sales data from Henkel and simulation data, both of which constitute structured time series datasets. The DNN algorithms employed for sales forecasting in this study include Long Short-Term Memory (LSTM), Neural Basis Expansion Analysis for Time Series (NBEATS), Temporal Convolutional Network (TCN), and Transformer. This thesis will contribute to sales forecasting by predicting future product sales while accounting for a large and highly diverse number of variables that can impact sales performance, as well as evaluating the performance of novel DL approaches to perform predictions in a sales forecasting context.

## 1.2 Research Objectives

The primary objective of this research is to leverage the proposed framework to develop innovative DNN models capable of accurately forecasting sales in the SCM domain at Henkel. To accomplish this objective, a comparative analysis is conducted among four DNN models based on their respective architectures. The performance of these models is evaluated using both real-world data (sales data) implementation and simulated data. Additionally, the forecast results from sales data are compared to the existing forecast utilized by Henkel, with the goal of creating a comprehensive global forecasting model applicable to all items and stores. Therefore, the research aims to achieve accurate sales forecasting by evaluating and comparing the performance of various DNN methods.

The second purpose of this study is to develop a global model that can be trained using a one-step process on all time series data, eliminating the need for additional processing. Given the large amounts of data involved, the research aims to make the model adaptable and efficiently trainable on large datasets, without requiring batch sizes that increase with the number of time series. In addition, the model incorporates additive coherence constraints along the edges of the hierarchy, which leads to better performance.

The application of this forecasting model will be deployed globally by an in-house data science team and steered centrally by a planning team at Henkel. As a result, this research offers valuable insights and recommendations for selecting the most suitable DNN model for sales forecasting in Henkel SCM based on the analysis of real sales data and experimental results. The long-term goal is reducing inventory, better order fulfilment, shorter cash-to-cash cycle times, higher profit margins, and minimizing the stock-out problems.

## 1.3 Problem Statements

Following the background, the research questions have been formulated to guide the achievement of the research objectives.

**RQ1**. How is the performance comparison between real-world datasets and simulated data by applying the four DNN algorithms?

**RQ2**. Which novel DL method produces the most accurate sales forecasts?

**RQ3**. How does the best DNN perform in comparison to the existing forecasting method in Henkel?

## 1.4 Related Work

This section presents a comprehensive review of literature related to the research topic of this final project. We cover the implementation of forecasting in the domains of global models, the forecasting methods used in several studies in terms of sales in retail or SCM, and the use of novel DL methods in diverse cases. Sales are commonly used as a proxy for demand in forecasting. While this is reasonable in most cases, particularly in supply chain cases, it is essential to understand that when stock-outs occur frequently or for extended periods of time, it is no longer an appropriate approximation since consumers with demand will be unable to purchase. Unless otherwise specified, the words sales and demand are used interchangeably throughout the rest of the argument. Because historical data on sales per time period is available, sales forecasting may be seen as a time series forecasting issue. Recently, sales forecasting has been approached in a number of different ways, and DL techniques are becoming increasingly popular due to their flexibility and ability to outperform traditional statistical approaches.

The M-competition, also known as the Makridakis Competitions, is a series of forecasting competitions initiated and organized by Professor Spyros Makridakis in the 1980s to compare the accuracy of various forecasting methods for time series data. The competitions were designed to encourage the development of new forecasting techniques and to evaluate their performance against existing methods (Makridakis et al., 2022a, 2018a). The M-competition significantly impacts on forecasting research, resulting in the development of novel approaches and the improvement of existing ones. The competitions also served as a benchmark for evaluating the accuracy of forecasting systems, which aided in the advancement of time series forecasting methods. The approaches of this competition have become common in large-scale industrial forecasting applications and have consistently been rated among the top entries in forecasting competitions (e.g., M4, M5, and M6).

The latest competition was M6 which had just finished January 2023, but in this literature review, we want to study the M5 competitions (Makridakis et al., 2022a,b) since the case study is similar to this research topic. According to Makridakis et al. (2022c, 2018b), up until now, forecasting models have been trained and optimized utilizing just the information provided in a single series, which is a series-by-series method. In cases where there is limited or sparse data or when series have a high correlation, cross-learning can greatly enhance forecasting accuracy by enabling the accurate prediction of individual series through learning from multiple series that is also called "global" model (Montero-Manso et al., 2020; Montero-Manso & Hyndman, 2021). The advantages of applying global models were emphasized in the M4 competition results (Smyl, 2020; Montero-Manso & Hyndman, 2021), where the winners used cross-learning approaches to extract meaningful information from the entire data set, and was followed by many other studies (Godahewa et al., 2021; Li et al., 2019). Montero-Manso & Hyndman (2021) stated that the benefits of using global models include developing forecasting algorithms for sets of time series that result in enhanced forecasting accuracy based on complexity analysis. This can be accomplished by including new characteristics as well as alternative model classes such as kernel approaches, deep networks, or regression trees. As the complexity of the local algorithm grows with the size of the set, the generalization bounds of the local and global algorithm can easily surpass the constant complexity. As a result, the existing empirical data based on DL has significant theoretical validity compared to the individually simple local method that has poorer generalization. These results are evidence that global models are a version of multi-task learning (Montero-Manso & Hyndman, 2021) and have antecedents in sequence-to-sequence models for forecasting (Mariet & Kuznetsov, 2019).

The M5 competition results support these findings and show that cross-learning is the general method of applying forecasting methodologies to increase overall forecast accuracy (Makridakis et al., 2022a). The goal of the M5 competition was to create the most accurate point forecasts for 42,840-time series representing the hierarchical unit sales of Walmart, the world's largest retail companies by revenue, as well as the most accurate prediction of the uncertainty regarding these forecasts (Makridakis et al., 2022b). The M5 winning approaches enhanced global models to effectively account for generally observable correlations across series of the same aggregate level and between series of different aggregation levels (Panagiotelis et al., 2021; Theodorou et al., 2022). Therefore, the global forecasting practice has long been considered to deal with such cases on retail or SCM where the challenges rely on extracting information from multiple time series at various aggregation levels, such as within store, product, product-category, product-department, product-store, and multiple countries as the data structure of M5 competition (Makridakis et al., 2022a). Instead of predicting series-by-series, the winners grouped those hierarchical structures into one global model that is a typical setup in a retail company. Besides, Spiliotis et al. (2020) examined retail sales forecasting using a data set of 3300 daily demand series and discovered that global forecasting strategies outperformed local approaches. Moreover, in the field of hierarchical forecasting on sales forecasting, Hyndman et al. (2011); Spiliotis et al. (2019) examined the base predictions produced at different

cross-sectional or temporal levels are often integrated or altered such that the final forecasts represent the patterns seen throughout the whole hierarchy while remaining coherent. Godahewa et al. (2021) research shows that with large datasets readily, global forecasting models that are trained across sets of time series often beat classic univariate forecasting models that work on isolated series.

By developing generic model structures that can easily handle complex non-linear patterns in data, artificial neural networks and ML have been shown to outperform the traditional statistical approaches (Sun et al., 2008; Loureiro et al., 2018; Kharfan et al., 2021). DNNs are known for their ability to capture complex features and multilevel representations from sales forecasting datasets (Weng et al., 2020). Weng et al. (2020) employed LSTM algorithm to predict the supply chain sales and assess its effectiveness and efficiency. LSTM is a deep learning technique that automatically extracts high-level temporal features from large datasets and accurately forecasts sales. The outcomes of the experiment demonstrate that the model presented by Weng et al. (2020) is capable of forecasting supply chain sales for a long-term period, making it suitable for application in industrial production environments.

The existing sales forecasting datasets comprise not only sequential data but also other data components, such as store and item information. Therefore, to incorporate this additional information, a spatiotemporal matrix must be generated. However, an Recurrent Neural Network (RNN) alone is insufficient to handle such datasets effectively. Convolutional Neural Network (CNN) is well-suited for processing such data as it can capture both scale-invariant features and local trend features (Wu et al., 2020; Ma et al., 2016). Laptev et al. (2017); Salinas et al. (2020) have demonstrated that RNN and CNN are capable of modelling intricate non-linear feature interactions and have achieved significant forecasting performance in situations where many related time series are present. Y. Chen et al. (2020) proposed TCN as a technique to learn complex patterns such as seasonality, holiday effects within and across series, and to enhance forecast accuracy, mainly when historical data is sparse or missing. TCN is a specialized type of CNN that is specifically designed for processing sequential data. TCN applies convolutions in the time domain, allowing it to learn temporal dependencies across multiple time steps (Bai et al., 2018). This makes TCN well-suited for modelling complex patterns such as seasonality and trend changes in time series data.

Hosseinnia et al. (2022) conducted a comprehensive systematic review of literature on the applications of DL in SCM. The study found that DL techniques have been widely used for sales forecasting due to their ability to effectively capture complex patterns and context-specific non-linear relationships between critical factors such as RNN and CNN. Novel DL algorithms are also emerging in the field of forecasting, such as NBEATS that was first introduced by Oreshkin et al. (2019). Several participants in M-competition (Makridakis et al., 2022a) used NBEATS algorithm as part of their solution, and one of them achieved the top rank in the competition by using an ensemble of NBEATS models with different configurations. The success of NBEATS in the M5 Competition and its ability to provide interpretable forecasts have made it a popular choice for time series forecasting tasks in various industries. Besides,

the transformer model introduced by Vaswani et al. (2017) and developed by Zeng et al. (2022) for long-term time series forecasting demonstrating the effectiveness in M-competition competing NBEATS achieved top results (Makridakis et al., 2022a).

## 1.5 Thesis Structure

The thesis is structured as follows: Chapter 2 provides the fundamental principles and concepts that are necessary to have a comprehensive understanding of the algorithms used in this study. Chapter 3 describes the methodology employed and the implementation of data validation techniques. Chapter 4 explains the implementation of each DNN algorithm using real-world datasets (sales data), including a comparative analysis of their performance. The performance results are further compared to the current forecasting method utilized by Henkel. In Chapter 5, the experiment results are presented, focusing on the simulation of data using the proposed DNN algorithms. This is followed by a conclusion and discussion points for further research in Chapter 6.

# Chapter 2

# Theoretical Background

## 2.1 Probability Theory

In probability theory, a random variable is a function that associates a real number with each element defined on a sample space. (Walpole et al., 1993). A probability distribution provides a framework for quantifying the likelihood of different possible outcomes of a random variable. The form of a probability distribution may differ depending on whether the random variable is discrete or continuous. Discrete random variables can only take on a finite or countably infinite set of possible values, while continuous random variables contains an infinite number of possibilities within a given range. A Probability Mass Function (PMF) is a function defined over the sample space of a discrete random variable $x$ which provides the probability associated with each specific value that $x$ is equal to a certain value. To define the random variable's probability coming within a distinct range of values, a Probability Density Function (PDF) is used as a function over the sample space $S$, where $S \subseteq \mathbb{R}$, of a continuous random variable $x$. The probability from random variable $x$ is within a certain interval can be obtained (Schervish & DeGroot, 2012).

There are several types of probability distributions depending on the nature of the problem being addressed. The most commonly used distribution over real numbers is the normal distribution, also known as the Gaussian distribution (Ross, 2014),

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \tag{2.1}$$

The normal distribution is characterized by two parameters: $\mu$ and $\sigma$, where $\mu$ determines the location of the central peak of the distribution and is equivalent to its mean. On the other hand, the parameter $\sigma$ is responsible for controlling the spread of the distribution, and $\sigma^2$ is equal to the variance of the distribution.

The exponential distribution is a continuous probability distribution that is frequently

used to describe the time elapsed between events (Walpole et al., 1993). There are situations where we need to use a probability distribution that has a concentrated peak at the point $x = 0$, which is useful for tasks such as generative modeling and anomaly detection. To accomplish this, we can use the exponential distribution with density given by,

$$p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x) \tag{2.2}$$

The exponential distribution uses the indicator function $\mathbf{1}_{x \geq 0}$ to assign probability zero to all negative values of $x$. Specifically, if we take the negative of the exponential distribution with rate parameter $\lambda$, we get a probability density function that is sharply peaked at $x = 0$. This is because the exponential distribution decays rapidly as $x$ increases, but increases rapidly as $x$ approaches 0. The parameter $\lambda$ is a measure of how quickly the probability density function decreases to zero. The mean of the distribution is $1/\lambda$, and the variance is $1/\lambda^2$.

## 2.2 Time Series Forecasting

According to Brockwell & Davis (2002), time series is a set of observations $y_t$, each being recorded at a specific time $t$. The observations are ordered sequentially with equal time intervals. Suppose there are $T$ periods of data available, with period $T$ being the most recent. We will let the observation on this variable at time period $t$ be donated by $y_t, t = 1, 2, \ldots, T$ (Montgomery et al., 2015). The observational data $y_t$ typically involves collecting an entire interval of time or at fixed time intervals. It can also represent a cumulative quantity such as the total demand for a product during period $t$. Different methods of time sampling require different approaches to data processing.

A general approach for statistical time series modelling is mentioned by Brockwell & Davis (2002); Montgomery et al. (2015) follows these steps: plot the time series and examine the primary aspects of the graph, particularly if there is a trend, a seasonal component, any obvious sharp changes in behavior, and any outlying findings; get stationary residuals by eliminating the trend and seasonal components; select a model to fit the residuals by estimating the unknown model parameters; validate the model to determine how it is likely to perform in the intended application by splitting the data into training, validation, and testing; deploy the forecasting model and invert the transformations performed in previous stages to return to the original series prediction.

Time series can be either univariate (containing a single variable at each point in time) or multivariate (including more than one variable at each point in time). A univariate time series is a series with a single time-dependent variable that are sequences of objects $o_1, o_2, \ldots, o_n$ declared at successive points $t_1, t_2, \ldots, t_n$ in time (Moritz et al., 2015). However, multivariate time series models include a number of variables that are both serially and cross-correlated, where each observation at a time $t$ is a vector

of values instead of a single value. The variables in the vector are typically closely interrelated for such series, which is considered to be a single observation in vector form.

A time series can be decomposed into a trend, seasonal, and residual component. The trend represents an overall increase (upward) or decrease (downward) in the value of the variable over time (Montgomery et al., 2015). Trends can be long-term or more dynamic and of shorter duration. The seasonal component repeats itself periodically on a regular basis, such as a year. The residuals are what is left after detrending and deseasonalizing the original data (Brockwell & Davis, 2002).

In statistics, each individual observation $y_t$ is viewed as a realization of a random variable $Y$ (Wheelwright et al., 1998). This means that it is important to select the right probability model for the data in order to make accurate predictions. Choosing the model with the best fit to historical data does not always result in a forecasting approach that generates the best forecasts of new data. Focusing too much on the model that generates the best historical fit frequently results in overfitting, or putting too many parameters or terms in the model merely to increase the model fit. In general, the optimal method is to choose the model with the minimum standard deviation of one-step-ahead forecast errors when applied to data that were not utilized in the fitting process (Montgomery et al., 2015). This is referred to as an out-of-sample forecast error standard deviation or Mean Squared Error (MSE). A common method for measuring out-of-sample performance is to use data splitting, which divides the time series data into two segments—one for model fitting and the other for performance testing. Cross-validation that will be further explained in Section 2.6 is another term for data splitting. The manner in which the data is separated is fairly arbitrary. Therefore, a reasonable rule of thumb is that the performance testing data set should include at least 20 or 25 observations.

Traditional time series methods are statistical techniques that have been used for many years to forecast time series data. Some of the common statistical traditional time series methods include: Naïve forecast, ARIMA (Autoregressive Integrated Moving Average), and Exponential Smoothing. The naïve forecasting methodology is the most fundamental approach to generating forecasts, and most often is found to be incredibly effective to be considered as the benchmark method for comparing models (Wheelwright et al., 1998). This method takes the observed value from the previous period and forecasts it for the upcoming period. This is frequently used as a baseline against which more advanced forecasting approaches are measured.

The ARIMA model can be comprehended by breaking down its components into the following categories: Autoregression (AR) which the model demonstrates a dynamic variable that is dependent on its own previous or lagged values; Integrated (I) represents the process of differencing the raw observations, which aids in making the time series stationary, thereby replacing the data values with the difference between them and their preceding values; and Moving Average (MA) where the component considers the correlation between an observation and an error residual that arises from a moving average model used on the lagged observations (Brockwell & Davis, 2002).

One of the most efficient and reliable forecasting methods for time series, which is also very popular in practice according to Weller & Crone (2012) is Simple Exponential Smoothing (sometimes also called "Single Exponential Smoothing"). It was first formulated by Brown & Meyer (1961) to forecast data with no clear trend or seasonal pattern. The forecast at time $t+1$ is equal to a weighted average between the most recent observation $y_t$ and the previous forecast $y_{t-1}$ that can be written as,

$$\hat{y}_{t+1|t} = \alpha y_t + (1-\alpha)\hat{y}_{t|t-1} \tag{2.3}$$

where $\alpha$ is the smoothing parameter, which is typically restricted within (0, 1) region (this region is arbitrary).

Let the first fitted value at time 1 be denoted by $l_0$ which we want to estimate. Hence, the process to calculate exponential smoothing can be explained as

$$\begin{aligned}
\hat{y}_{2|1} &= \alpha y_1 + (1-\alpha)\, l_0 \\
\hat{y}_{3|2} &= \alpha y_2 + (1-\alpha)\hat{y}_{2|1} \\
\hat{y}_{4|3} &= \alpha y_3 + (1-\alpha)\hat{y}_{3|2} \\
&\vdots \\
\hat{y}_{t-1} &= \alpha y_{t-1} + (1-\alpha)\hat{y}_{t-1|t-2} \\
\hat{y}_{t+1|t} &= \alpha y_t + (1-\alpha)\hat{y}_{t|t-1}
\end{aligned} \tag{2.4}$$

Substituting each equation into the following equation, we obtain

$$\begin{aligned}
\hat{y}_{3|2} &= \alpha y_2 + (1-\alpha)[\alpha y_1 + (1-\alpha)l_0] \\
&= y_2 + \alpha(1-\alpha)y_1 + (1-\alpha)^2 l_0 \\
\hat{y}_{4|3} &= \alpha y_3 + (1-\alpha)[\alpha y_2 + \alpha(1-\alpha)y_1 + (1-\alpha)^2]l_0 \\
&= \alpha y_3 + (1-\alpha)[\alpha y_2 + \alpha(1-\alpha)^2 y_1 + (1-\alpha)^3]l_0 \\
&\vdots \\
\hat{y}_{t+1|t} &= \sum_{j=0}^{t-1} \alpha(1-\alpha)^j y_{t-j} + (1-\alpha)^t l_0
\end{aligned} \tag{2.5}$$

The last term becomes small for large $t$, such that the weighted average form leads to the same forecast equation 2.3. The smoothing parameter $\alpha$ is typically interpreted as a weight between the most recent actual value and the one-step-ahead predicted one. If the smoothing parameter is close to zero, the prior fitted value $\hat{y}_t$ has more weight and the new information is ignored. If $\hat{\alpha} = 0$, then the method becomes equivalent to the global mean method. When it is close to one, then most of the weight is assigned to the actual value $y_t$. If $\hat{\alpha} = 1$, the method transforms into Naïve method. By modifying the smoothing parameter value, we can decide how to approximate the data and filter out the noise (Hyndman & Athanasopoulos, 2018).

## 2.3 Deep Learning Basics

This section provides the fundamental principles underlying deep learning. In order to grasp the concepts of DL effectively, it is essential to possess a solid foundation in the basic principles of ML, as DL is a specialized subset of ML (Goodfellow et al., 2016). According to Bishop & Nasrabadi (2006) ML is a broad field that includes a variety of algorithms and techniques that allow machines to learn from data and make predictions or decisions without being explicitly programmed. It involves using statistical models and algorithms to analyze and find patterns in data, and uses those patterns to make predictions or decisions about new data. The goal of ML is to build systems that can automatically improve their performance with experience, and ultimately to develop machines that can learn, reason, and act like humans. Therefore, applying ML allows a system to learn from problem-specific training data to automate the process of analytical model building and solve associated tasks (Janiesch et al., 2021).

Based on the given problem and the available data, we can distinguish three types of ML; supervised learning, unsupervised learning, and reinforcement learning. Supervised learning involves using a training dataset consisting of input examples and labeled output values to calibrate the parameters of a machine learning model (Janiesch et al., 2021). Once the model has been successfully trained, it can be used to predict the target variable given new input data. On the other hand, unsupervised learning algorithms are used to identify the underlying structure of a dataset containing multiple features. In the context of DL, the goal is often to learn the probability distribution that generated the dataset, either explicitly, such as in density estimation, or implicitly, for tasks such as synthesis or denoising. Clustering is another unsupervised learning algorithm that divides the dataset into clusters of similar examples (Goodfellow et al., 2016). Meanwhile, reinforcement learning algorithms involves the model interacting with an environment and receiving feedback in the form of rewards or punishments. Instead of providing input and output pairs, reinforcement learning involves specifying a current state of the system, a goal, and a list of allowable actions and their environmental constraints. The ML model then experiences the process of achieving the goal by itself using trial and error to maximize the reward (Janiesch et al., 2021).

Figure 2.1 depicted that deep learning is a subfield of machine learning that involves training artificial neural networks with multiple layers (deep neural networks) to perform complex tasks such as image recognition, natural language processing, and speech recognition (Goodfellow et al., 2016). According to Janiesch et al. (2021) DNN specifically refers to the architecture of neural networks with multiple hidden layers, DL involves a wider range of methods that are used to optimize the performance of these networks in various applications.

It is widely believed that DL can overcome the ML problems that is the curse of dimensionality. This phenomenon occurs when the number of dimensions in the data is high, making many ML problems exceedingly difficult. When the number of variables increases, the number of possible distinct configurations of those variables

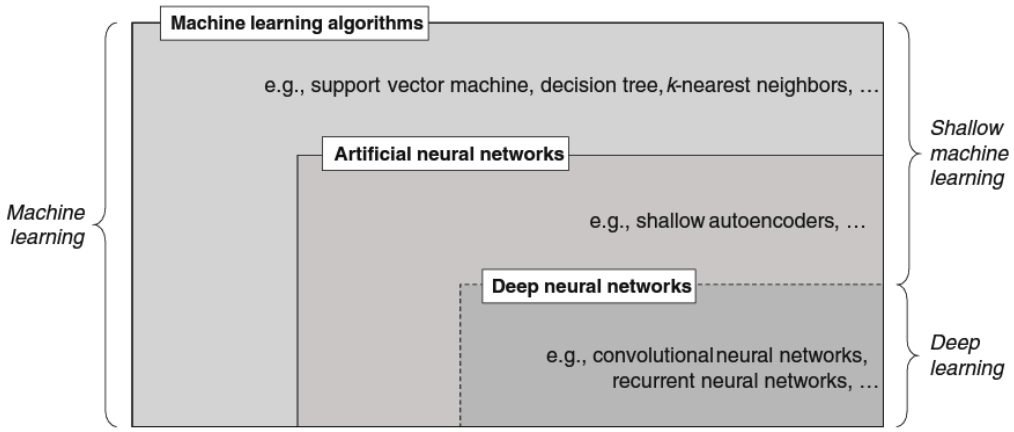


Figure 2.1: Venn Diagram of Machine Learning and Deep Learning Concepts

grows exponentially. This poses a statistical challenge since the number of possible configurations is much larger than the number of training examples.

DL algorithms are particularly well-suited for complex tasks such as image and speech recognition, complex time series, natural language processing, and decision making. They can automatically extract relevant features from raw data, and can learn and improve over time, making them particularly powerful for solving problems that would be difficult or impossible for humans to solve on their own. According to Schmidhuber (2015), the primary objective of DNN is to receive a given set of inputs and perform successive calculations on them to generate outputs that can address real-world problems, such as sequential data in time series and classification tasks. Typically, DNN consists of an input layer, an output layer, and a sequential flow of data within a deep network architecture. To extract high-level functions from input data, DNN employs multiple layers of nodes. This enables the network to address problems in a more comprehensive manner, allowing it to make informed conclusions or predictions based on the available information and the desired outcome.

## 2.4 Gradient Descent

In gradient-based numerical optimization algorithms, the computation of parameter updates is mainly based on the gradient or an estimator of the gradient. Online or Stochastic Gradient Descent (SGD), which is a widely used optimization algorithm, updates the parameters after each example is processed (Bengio, 2012). The gradient is essentially a vector that points in the direction of the steepest ascent of a function, and its estimation is a critical step for optimizing the performance of machine learning models. SGD computes a noisy gradient estimate using a randomly selected subset of the training data, which is usually more computationally efficient than processing the entire dataset at once. By updating the model parameters using the gradient information, the algorithm aims to iteratively optimize the model's performance on the training data.

$$\theta^{(t)} \leftarrow \theta^{(t-1)} - \epsilon_t \frac{\partial L(z_t, \theta)}{\partial \theta} \tag{2.6}$$

where $z_t$ is an example sampled at iteration $t$ and where $\epsilon_t$ is a hyperparameter that is called the learning rate. The choice of the learning rate is an important hyperparameter since it determines the size of the update steps taken during the training process. If the learning rate is set too large, the algorithm may overshoot the optimal point and the average loss function may increase instead of decreasing, resulting in poor model performance Bengio (2012). Otherwise, if the learning rate is too small, it may lead to overfitting and may not adequately represent the true underlying patterns and variations present in the data. This can lead to biased and incomplete learning, resulting in suboptimal models that fail to capture the complexity of the problem. Therefore, it is important to note that the true gradient direction obtained by averaging over the entire training set, represents the locally steepest descent direction, it may not necessarily indicate the correct direction when considering larger steps.

## 2.5 Hyperparameter Sets

A hyperparameter is a variable that is set prior to applying the learning algorithm to data and is not selected by the algorithm itself. The hyperparameters can be manually fixed or tuned by an algorithm, but their values must be selected. The basic concept of hyperparameter setting is to find the optimal set of hyperparameters that can minimize the error on the validation set and improve the performance of the model. Hyperparameters play a crucial role in DL and can significantly impact the performance of the model, which are settings that are not learned during the training process but are instead set before training. Examples of hyperparameters include the learning rate, regularization strength, batch size, and number of hidden units (Bengio, 2012).

Bergstra & Bengio (2012) proposes a method called "random search" for hyperparameter optimization. The basic concept is to randomly sample hyperparameters from a specified search space and evaluate their performance on a validation set. This process is repeated for a certain number of iterations or until a stopping criterion is met. By using random search, the authors aim to find good hyperparameter values without making any assumptions about the structure of the search space or the relationship between hyperparameters. They argue that random search is more effective and efficient than grid search, which is a commonly used method for hyperparameter optimization that requires a pre-defined grid of hyperparameters to be searched.

Bengio (2012) recommends a systematic approach to setting hyperparameters, such as random search or Bayesian optimization. They suggest setting a range for each hyperparameter and randomly sampling from this range to create a set of hyperparameters. The model is then trained and evaluated for each set of hyperparameters to determine which set performs best. This process can be computationally expensive

but is essential to ensure that the model is optimized correctly.

A learning algorithm takes training data as input and produces a predictor or model as output. In deep learning, there are many hyperparameters to be set, which can make it difficult to adjust them all manually. The selection of hyperparameter values is equivalent to model selection, that is how to choose the most appropriate learning algorithm from a set of options. The performance of the algorithm on its training data can be used to select the values of some hyperparameters, but most cannot be selected this way. For hyperparameters that impact the effective capacity of a learner, it makes more sense to select their values based on out-of-sample data. Once some out-of-sample data has been used to select hyperparameter values, it cannot be used to obtain an unbiased estimator of generalization performance, so the generalization error of the pure learning algorithm (with hyperparameter selection hidden inside) is typically estimated using a test set (or double cross-validation for small datasets).

### 2.5.1 Neural Network Hyperparameters

Various learning algorithms have various sets of hyperparameters, and it is useful to understand the sorts of selections that practitioners must make when deciding on their values, particularly in applicable neural networks and DL techniques. To understand learning algorithms, we can break them down into two parts - the training criterion and the model (which includes a family of functions or a parametrization), and the procedure used to optimize the criterion. This means there are hyperparameters associated with the optimizer, and those associated with the model itself - including the function class, regularizer, and loss function. It is important to distinguish between the two to effectively optimize the algorithm (Bengio, 2012). The following list is a comprehensive description of hyperparameters for those DL methods mainly used in SGD.

- The **initial learning rate** ($\epsilon_0$) is one of the most important hyperparameter which inputs have been standardized or mapped to the interval (0,1), with typical values for the learning rate fall between less than $1$ and greater than $10^{-6}$. However, these values should not be considered fixed ranges as they greatly depend on the model's parametrization. Although a default value of $0.01$ usually works for standard multi-layer neural networks, relying solely on this default value would be unwise. If only one hyperparameter can be optimized and SGD is being used, tuning the learning rate is the most crucial hyperparameter to optimize.

- The choice of the method to reduce or adjust the **learning rate schedule**, along with its hyperparameters such as the time constant $\tau$, is an important consideration in training a neural network. By default, a constant learning rate is used over the course of training, with a large value for $\tau$ for every iteration. While it is typically not necessary to deviate from this default, there are cases where using a non-default learning rate schedule can be beneficial. One such

example is the $O(1/t)$ learning rate schedule, used in Bergstra & Bengio (2012) is

$$\epsilon_t = \frac{\epsilon_o \tau}{\max(t, \tau)} \tag{2.7}$$

which maintains a constant learning rate for the first $\tau$ iterations and then decreases of the order $O(1/t^{\alpha})$, with a recommended value of $\alpha = 1$ based on traditional analysis of convex optimization problems. Moulines & Bach (2011) suggested that smaller values of $\alpha$ should be used in the non-convex case ($\alpha \leq 1$), especially when using a gradient averaging or momentum technique. The default value for $\tau$ is usually infinity, meaning that the learning rate $\epsilon_t$ remains constant over training iterations from epoch to epoch. However, in some cases, it may be beneficial to choose a different value for $\tau$. One heuristic method for setting $\tau$ is to keep the learning rate constant until the training criterion stops decreasing significantly, determined by a relative improvement threshold. Another approach is to use an adaptive learning rate heuristic, such as the one proposed by Bottou (2010), which involves training with $N$ different learning rates in parallel and selecting the value that gives the best results until the next re-estimation of the optimal learning rate. This method is performed at regular intervals during training using a small subset of the training set (Bengio, 2012).

- The **mini-batch size**, denoted by $B$, is a crucial hyperparameter in deep learning. The choice of $B$ varies depending on the task, dataset, and available computational resource. A common range for $B$ is between one and a few hundred, with a default value $B = 32$ in many cases. Larger values of B, above 10, can leverage the computational speed-up gained from matrix-matrix products over matrix-vector products. The influence of mini-batch size $B$ primarily affects the computational aspect of training. With larger values of $B$, computation is faster (provided proper implementation) but requires more example visits to achieve the same error since fewer updates occur per epoch. According to theory, this hyperparameter should impact training time rather than test performance. Hence, it can be optimized independently of other hyperparameters by comparing training curves, such as training and validation error against the amount of training time, after selecting other hyperparameters except for the learning rate. It is possible for $B$ and $\epsilon_0$ to have a slight interaction with other hyperparameters. Therefore, it is recommended to re-optimize both at the end of the optimization process. Once $B$ has been selected, it can be kept fixed while the other hyperparameters are further optimized, except for the momentum hyperparameter if it is used (Bengio, 2012).

- **Number of training iterations** $T$ which is measured in mini-batch updates. This hyperparameter has a unique characteristic as it can be optimized without additional costs using the principle of early stopping. By monitoring the out-of-sample error, estimated on a validation set, as the training progresses (at regular intervals), one can determine the optimal training time for a given setting of all the other hyperparameters. Early stopping is a cost-effective way to prevent severe overfitting, even when other hyperparameters may lead

to overfitting. It also conceals the overfitting effects of other hyperparameters, which may obscure the analysis of individual effects. Early stopping can balance out the performance achieved by various overfitting hyperparameter setups by compensating for high capacity with a shorter training period.

- **Momentum** $\beta$ has been used to smooth out the stochastic gradient samples produced by SGD. While a default value of $\beta = 1$ (no momentum) is sufficient in many cases, momentum has shown to have a positive impact in certain situations. According to Bottou (2010), for very large training sets, it may be impossible to achieve better rates than those obtained with ordinary SGD. However, the constant in front of the rate can still be significantly reduced by using second-order information online.

- **Layer-specific optimization hyperparameters** although rarely done, it is feasible to employ different values of optimization hyperparameters, including the learning rate, for individual layers in a multi-layer network, although this is not commonly implemented. This technique is especially suitable in the context of layer-wise unsupervised pre-training, as each layer can be trained separately while maintaining the lower layers (Bengio, 2012)

### 2.5.2 Hyperparameters of the Model and Training Criterion

In DL, it is important to carefully choose and tune the hyperparameters of a model and training criterion in order to obtain the best performance. This typically involves a process of trial and error, where different hyperparameter settings are tried and evaluated on a validation set, and the best-performing hyperparameters are selected.

- **Number of hidden units** $n_h$. The size of each layer in a multi-layer neural network is usually adjustable and affects the model's capacity. To ensure generalization performance, it is crucial to set the layer size ($n_h$) to be sufficiently large, given the use of early stopping and other regularization techniques (such as weight decay). Even if $n_h$ is set too large, it usually has little impact on generalization performance. However, larger layer sizes require a proportional increase in computation, typically in $O(n_h^2)$ when scaling all layers at the same time in a fully connected architecture.

- **Weight decay** regularization coefficient $\lambda$. One method to prevent overfitting in machine learning models is to include a regularization term in the training criterion, which restricts the capacity of the model. This term can push the model's parameters towards a prior value, usually zero. L2 regularization includes a term $\lambda \Sigma_i \theta_i^2$ in the training criterion, while L1 regularization includes a term $\lambda \Sigma_i |\theta_i|$. Both types of regularization terms can be used, and the regularization strength is controlled by the regularization coefficient $\lambda$. There exists a sound Bayesian justification for the regularization term, where it is considered as the negative log-prior $-\log P(\theta)$ on the parameters $\theta$. The training criterion is then the negative joint likelihood of data and parameters,

$-\log P(data, \theta) = \log P(data|\theta) - \log P(\theta)$, where the loss function $L(z, \theta)$ is interpreted as $-\log P(z, \theta)$ and $-\log P(data|\theta) = -\sum_{t=1}^{T} L(z, \theta)$ if the data comprises of $T$ identically and independently distributed (i.i.d) examples $z_t$. It is crucial to note that in stochastic gradient-based learning, it is advisable to use an unbiased estimator of the gradient of the overall training criterion, which includes both the total loss and the regularizer. However, only a single mini-batch or example is considered at a time. There are two reasons for treating output weights differently than other weights in a neural network. Firstly, we can restrict the network's capacity by applying regularization solely to the output weights, which eliminates the need to rely solely on early stopping. Secondly, the inputs and outputs may be sparse, which necessitates a different treatment of the output weights compared to the hidden units (Bengio, 2012).

- **Sparsity of activation** regularization coefficient $\alpha$. A prevalent approach in the DL field, as described by Goodfellow et al. (2012), involves introducing a regularization term into the training objective that promotes sparsity in the hidden units. This sparsity constraint aims to encourage the hidden units to have values that are close to 0 or exactly 0. Sparse representations are often considered advantageous since they promote representations that disentangle the underlying factors of representation. It is worth noting that increased sparsity can be offset by incorporating more hidden units in the network. Various methods have been proposed to encourage the activation of hidden units to be sparse or close to zero. Le et al. (2011) proposed a method of penalizing the L1 norm of the representation or another function of the hidden units' activation. However, this approach may not work well with non-linearities such as the hyperbolic tangent, which saturate around -1 and 1, rather than around 0. Alternatively, it is possible to penalize the biases of the hidden units, pushing them towards negative values (Goodfellow et al., 2009).

- **Neuron non-linearity**. The typical neuron output is $s(a) = s(w'x+b)$, where $x$ is the vector of inputs into the neuron, $w$ the vector of weights and $b$ the offset or bias parameter, while $s$ is a scalar non-linear function. The most commonly used activation functions are the sigmoid $1/(1+e^{-a})$; the hyperbolic tangent $\frac{e^a-e^{-a}}{e^a+e^{-a}}$; the rectifier max(0,a) and the hard $tanh$ (Collobert & Bengio, 2004).

- **Weights initialization scaling coefficient**. Biases can generally be set to zero initially, but weights should be initialized carefully to avoid symmetries among hidden units in the same layer. Deciding whether to use unsupervised pre-training and which algorithm to use for unsupervised feature learning is a crucial decision. In most cases, unsupervised pre-training has been found to be helpful and seldomly to be harmful, but it does require additional training time and the specification of extra hyperparameters (Bengio, 2012).

- **Random seeds**. In general, the choice of a random seed has a minimal impact on the results and can be disregarded for most of the hyperparameter search processes. However, if computing resources are available, running a final set of

jobs with different random seeds (typically 5 to 10) for a small set of best choices of hyperparameter values may lead to a slight improvement in performance.

- **Preprocessing**. A commonly used nonlinear preprocessing technique recommended by Mesnil et al. (2012) is the uniformization of features. This method estimates the cumulative distribution of each feature, denoted as $F_i$, and transforms each feature value $x_i$ using its corresponding quantile, $F_i^{-1}(x_i)$. This transformation produces a normalized rank or quantile for the value of $x_i$. Another simple transformation is to apply a nonlinear function such as the logarithm or square root to the input features, which can help reduce the tails of the distributions and make them more Gaussian-like.

## 2.6 Cross-Validation

Certain settings, referred to as hyperparameters, are not learned through the gradient descent algorithm. This is because optimizing these hyperparameters directly using gradient descent can be challenging or computationally expensive. In such cases, a cross-validation set is used to evaluate the hyperparameters which the algorithm did not observe during the training process. This helps in avoiding overfitting of the model to the training data and ensures that the model is generalizable to new, unseen data. By tuning the hyperparameters using the cross-validation set, we can find the optimal settings that result in the best performance of the model on the test set. The process of selecting the best hyperparameters is also known as hyperparameter tuning, which can be done using various methods such as grid search, random search, and Bayesian optimization (Goodfellow et al., 2016).

To make proper choices about the model, including its hyperparameters, a standard approach is to split the data into two parts. One subset is used for learning the model parameters, while the other subset is used as a validation set for estimating the generalization error during or after training. The validation set helps in updating the hyperparameters accordingly. The subset of data used to learn the parameters is still commonly called the training set, even though it may cause confusion with the larger pool of data used for the entire training process. On the other hand, the subset of data used to guide the selection of hyperparameters is called the validation set. Typically, we use around 80 percent of the training data for training and 20 percent for validation. However, since the validation set is used to "train" the hyperparameters, it may underestimate the generalization error. Nonetheless, it usually underestimates the generalization error by a smaller amount than the training error does. After hyperparameter optimization is complete, the generalization error can be estimated using the test set.

The process of dividing a dataset into a fixed training set and test set can be problematic, particularly if the test set is small. A small test set results in statistical uncertainty around the estimated average test error, which can make it difficult to claim that one algorithm works better than another on the given task. In cases where

the dataset is small, alternative procedures are available that enable the use of all examples in the estimation of the mean test error, at the cost of increased computational complexity. One such procedure is $k$-fold cross-validation, which involves partitioning the dataset into $k$ non-overlapping subsets, and estimating the test error by taking the average across $k$ trials. In each trial, the $i$-th subset of the data is used as the test set, and the rest of the data is used as the training set (Grandvalet & Bengio, 2004).

## 2.7 Overfitting and Underfitting

The training and test data are generated by a probability distribution over data sets called the data generating process. To study the relationship between training error and test error, we assume that the individual samples are i.i.d. These assumptions imply that the examples in each data set are not related to each other, and that the training set and test set have the same statistical properties, i.e., they are drawn from the same probability distribution. By making these assumptions, we can describe the data generation process using a single probability distribution that generates every example in both the training and test sets. This probabilistic framework allows us to study the relationship between the training error and test error in a mathematical way (Goodfellow et al., 2016).

One important observation is that the expected training error of a randomly selected model is equal to the expected test error of that model. This is because both expectations are formed using the same data set sampling process, where we repeatedly sample from the probability distribution $p(x, y)$. However, when using an ML algorithm, we sample the training set and use it to choose the parameters to reduce training set error, then sample the test set. In this case, the expected test error is greater than or equal to the expected value of training error.

In practical ML applications, we do not set the parameters of the algorithm beforehand and then sample the data sets. Rather, we sample the training set and use it to adjust the algorithm parameters to minimize the training error. Then, we sample the test set to evaluate the performance of the algorithm. Noting that under this process, the expected test error is always greater than or equal to the expected training error. Therefore, it is crucial to develop ML algorithms that are effective at minimizing both the training error and the gap between the training and test error. The primary goals in assessing its overall performance are to minimize training error and the gap between training and test error.

Therefore, the goal of a good ML algorithm is to have low training error while generalizing well to new, unseen test data. Those two factors correspond to the two central challenges in machine learning as well as deep learning: underfitting and overfitting. Underfitting occurs when the model is not able to achieve a sufficiently low error value on the training set. Meanwhile overfitting occurs when the model performs very well on the training data, but it does not generalize well to the test data. In

other words, overfitting happens when the difference between the error values on the training and test sets is too large (Goodfellow et al., 2016). Once overfitting happens, there are two general strategies to overcome as mentioned by James et al. (2013): Slow Learning and Regularization. The Slow Learning approach involves gradually fitting the model in an iterative manner by utilizing gradient descent. The fitting procedure is then halted when the signs of overfitting due to gradient descent are detected. The second solution is regularization which involves adding a penalty term to the objective function that the model is trying to minimize. This penalty term discourages the model from learning complex, high-variance relationships between the features and the target variable. In regularization, we modify the learning algorithm that is intended to reduce its generalization error but not its training error.

There are two common types of regularization: L1 regularization and L2 regularization. L1 regularization adds a penalty term to the objective function that is proportional to the absolute value of the model weights. This penalty term encourages the model to learn sparse weights, where many of the weights are exactly zero. This has the effect of reducing the number of features that the model uses, which can help to prevent overfitting. L2 regularization, on the other hand, adds a penalty term that is proportional to the square of the model weights. This penalty term encourages the model to learn small weights, which has the effect of smoothing the decision boundary and reducing the sensitivity of the model to small changes in the input data.

# Chapter 3

# Methodology

## 3.1 Long Short-Term Memory

RNN is a family of neural networks for processing sequential data $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$, which allow cyclical connections in the network (Rumelhart et al., 1986). RNN possesses the ability to handle much longer sequences than non-recurrent networks due to their specialization in sequential data processing. Moreover, these networks can efficiently process sequences of varying lengths over time (Goodfellow et al., 2016).

RNNs have a crucial advantage in utilizing contextual information for mapping input and output sequences, by allowing connections between hidden units with a time delay (Graves & Graves, 2012). This unique structure enables the model to retain information from the past and discover temporal relationships between distant events in the data. However, despite its simple yet powerful design, recurrent networks are challenging to train due to the potential exponential decay or explosion of input influence on the hidden layer and network output as it cycles through the recurrent connections. This issue is commonly known as the vanishing gradient and exploding gradient problems, which have been discussed by Bengio et al. (1994); Hochreiter et al. (2001).

The approach favored to tackle those problems is the LSTM method (Hochreiter & Schmidhuber, 1997). The LSTM architecture consists of a set of recurrently connected subnets, known as memory blocks that can capture the long-term dependencies of the data. Each block contains one or more self-connected memory cells and three multiplicative units (Graves & Graves, 2012). The multiplicative gates allow LSTM memory cells to store and access information over long periods of time, thereby mitigating the vanishing gradient problem. This memory cell is controlled by three gates: an input gate, an output gate, and a forget gate. The input gate determines how much new information should be stored in the memory cell, based on the current input and the previous output. The forget gate decides how much of the old information should be discarded from the memory cell. The output gate determines how much of the memory cell's contents should be output to the next layer or time step. By

selectively adding or removing information from the memory cell, LSTM can effectively maintain relevant information over long sequences, while filtering out noise and irrelevant information.

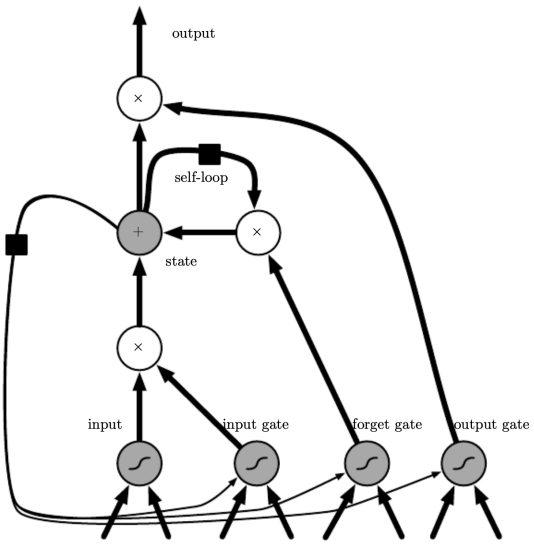


Figure 3.1: Block Diagram of the LSTM

Figure 3.1 shows the architecture of the LSTM within RNN cell, which is designed to address the vanishing gradient problem in conventional recurrent networks (Goodfellow et al., 2016). Unlike ordinary recurrent networks that use hidden units, cells are recurrently connected to each other. The input feature is computed using a regular artificial neuron unit, and its value is accumulated into the state if the sigmoidal input gate permits. The state unit contains a linear self-loop whose weight is regulated by the forget gate. The output gate can disable the output of the cell. All gating units employ a sigmoid nonlinearity, while the input unit can use any squashing nonlinearity. The state unit can also serve as an additional input to the gating units. The black square signifies a delay of a single time step.

Leaky units enable the network to gather information such as evidence for a certain characteristic or category over time. However, once that information is used, it may be beneficial for the neural network to forget the previous state (Pascanu et al., 2013). In addition to the RNN's outside recurrence, LSTM recurrent networks contain "LSTM cells" with an inside recurrence (a self-loop). Each cell has the same inputs and outputs as a traditional recurrent network, but it includes extra parameters and a gating system that regulates the flow of information. The state unit $s_i^{(t)}$, which possesses a linear self-loop like the leaky $i$ units mentioned in the preceding section, is the most significant component. The self-loop weight is controlled by a forget unit $f_i^{(t)}$ for time step $t$ and cell $i$, that sets this weight to a value between 0 and 1 via a sigmoid unit as follows.

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{3.1}$$

where $\boldsymbol{x}^{(t)}$ refers to the current input vector and $\boldsymbol{h}^{(t)}$ is the current hidden layer vector containing the outputs of all LSTM cells. Moreover, $\boldsymbol{b}^f, \boldsymbol{U}^f, \boldsymbol{W}^f$ are respectively biases, input weights, and recurrent weights for the forget gates. Then, the LSTM cell internal state is updated with a conditional self-loop as follows.

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left( b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right) \tag{3.2}$$

where $\boldsymbol{b}, \boldsymbol{U}, \boldsymbol{W}$ are respectively biases, input weights, and recurrent weights into LSTM cell. The external input gate unit $g_i^{(t)}$ is computed similarly to the forget gate with a sigmoid unit to obtain a gating value between 0 and 1, but with its own parameters that can be written as.

$$g_i^{(t)} = \sigma \left( b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right) \tag{3.3}$$

The LSTM cell's output $h_i^{(t)}$ may likewise be turned off using the output gate $q_i^{(t)}$ which similarly employs a sigmoid unit for gating:

$$h_i^{(t)} = \tanh(s_i^{(t)})\, q_i^{(t)} \tag{3.4}$$

$$q_i^{(t)} = \sigma \left( b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right) \tag{3.5}$$

which contains parameters $\boldsymbol{b}^o, \boldsymbol{U}^o, \boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively.

Rectified Linear Unit (ReLU) activation function is employed in this study, which can be expressed as $f(x) = \max(0, x)$. ReLU contributes to the enhancement of neural networks by expediting the training process. The computation of gradients becomes straightforward, resulting in values of either 0 or 1 based on the sign of $x$. Furthermore, the computational step involved in ReLU is not complicated: negative elements are simply replaced with 0, eliminating the need for exponentials, multiplication, or division operations (Niu et al., 2020; Krizhevsky et al., 2017):

## 3.2 Neural Basis Expansion Analysis for Time Series

The NBEATS architecture is a DNN that uses backward and forward residual links and a very deep stack of fully-connected layers to perform time series forecasting.

It diverges from other deep learning frameworks for time series forecasting in several fundamental ways. Firstly, the fundamental architecture is designed to be basic and universal while also expressive, which enables the exploration of pure DL architectures in time series prediction. Secondly, the architecture does not rely on time series-specific feature engineering or input scaling. Finally, the architecture can be extended towards making human interpretable results, which is necessary for investigating interpretability.

Oreshkin et al. (2019) published NBEATS with a simple, general, and expressive architecture that does not rely on time-series-specific feature engineering or input scaling. The basic idea behind the architecture is to use a set of basis functions to model the time series data. These basis functions are learned by the network during training, and they are combined in different ways to produce the final forecast. The network uses a stack of fully connected layers to learn how to combine these basis functions in an optimal way. Therefore, NBEATS can be trained on multiple time series, each one representing a different distribution. Because there are no recurrent or self-attention layers in the model, training is faster and gradient flow is stable.

Consider a length-$H$ forecast horizon a length-$T$ observed series history $[y_1, \ldots, y_T] \in \mathbb{R}^T$ with lookback window of length $t \leq T$ ends with $y_T$. We want to predict the future values $\boldsymbol{y} \in \mathbb{R}^H = [y_{T+1}, y_{T+2}, \ldots, y_{T+H}]$, which the forecast of $\boldsymbol{y}$ denoted by $\hat{\boldsymbol{y}}$. The last observed value $y_T$ serve as model input, given by $\boldsymbol{x} \in \mathbb{R}^t = [y_{T-t+1}, \ldots, y_T]$.

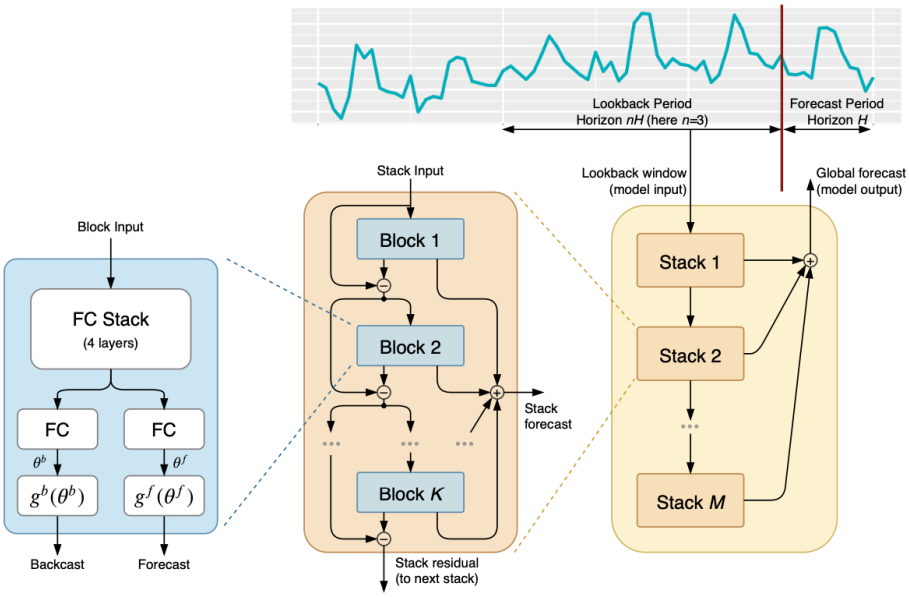


Figure 3.2: The Architecture of NBEATS

The proposed deep architecture consists of a basic building block, which is replicated throughout the network (depicted in blue in Figure 3.2). Each block that is presented by $l$ contains one input $\boldsymbol{x}_l$ and two outputs ($\hat{\boldsymbol{x}}_l$ and $\hat{\boldsymbol{y}}_l$), referred to as backcast and forecast. The overall model input that represents $\boldsymbol{x_l}$ for the very first block in the model is a historical lookback window of a particular length end with the last

measured observation. Considering the length of input window to be a multiple of the forecast horizon $H$, and typical lengths of $\boldsymbol{x}$ in a range from $2H$ to $7H$. The remaining block inputs $x_l$ are residual outputs of the previous blocks. Similar to the output from the very first block, the rest of the blocks produce two outputs; $\hat{\boldsymbol{y}}_l$ for the block's forward forecast of length $H$ and $\hat{\boldsymbol{x}}_l$ for the block's backcast estimate of $\boldsymbol{x}_l$ by considering the constraints of the functional space that the block is allowed to employ for approximating signals.

The global model output is generated as the sum of all the forecasts from the individual blocks. The backcast generates a set of historical embeddings from the previous inputs that are used for the following block. This mechanism ensures that the subsequent input does not contain the portion that has already been predicted by the previous block. As a result, the following blocks can concentrate on the remaining unexplained parts of the input data. The forecast then generates a set of future embeddings that are used to make predictions about future values in the time series.

The block architecture is basically comprised of two parts. The first part is a fully connected network that produces the forward $\theta_l^f$ and the backward $\theta_l^b$ predictors of expansion coefficients. As we can see at Figure 3.2, the block index $l$ dropped for $\theta_l^b, \theta_l^f, g_l^b, g_l^f$. The second part comprises of the backward $g_l^b$ and the forward $g_l^f$ basis layers that accept the respective forward $\theta_l^f$ and backward $\theta_l^b$ expansion coefficients, project them internally on the set of basis functions and produce the backcast $\hat{\boldsymbol{x}}_l$ and the forecast outputs $\hat{\boldsymbol{y}}_l$.

The following equations describe the functioning of the first section of the $l$-th block:

$$\begin{aligned}
\boldsymbol{h}_{l,1} &= \mathsf{FC}_{l,1}(\boldsymbol{x}_l), \\
\boldsymbol{h}_{l,2} &= \mathsf{FC}_{l,2}(\boldsymbol{h}_{l,1}), \\
\boldsymbol{h}_{l,3} &= \mathsf{FC}_{l,3}(\boldsymbol{h}_{l,2}), \\
\boldsymbol{h}_{l,4} &= \mathsf{FC}_{l,4}(\boldsymbol{h}_{l,3}). \\
\theta_l^b &= \mathsf{LINEAR}_l^b\,(\boldsymbol{h}_{l,4}), \\
\theta_l^f &= \mathsf{LINEAR}_l^f\,(\boldsymbol{h}_{l,4}).
\end{aligned} \tag{3.6}$$

LINEAR layer is a linear projection layer which means $\theta_l^f = \boldsymbol{W}_l^f\,\boldsymbol{h}_{l,4}$ and the FC layer is a standard fully connected layer with ReLU layer. Therefore, for $\mathsf{FC}_{l_1}$ we have $\boldsymbol{h}_{l_1} = \mathsf{ReLU}(\boldsymbol{W}_{l,1}\boldsymbol{x}_l + \boldsymbol{b}_{l,1})$. This section of the architecture is responsible for predicting the forward expansion coefficients $\theta_l^f$ with the ultimate objective of optimizing the accuracy of the partial forecast $\hat{\boldsymbol{y}}$ by appropriately combining the basis vectors given by $g_l^f$. Furthermore, the subnetwork estimates backward expansion coefficients $\theta_l^b$ which are utilized by $g_l^b$ to construct an estimate of $\boldsymbol{x}_l$ with the ultimate objective of assisting the downstream blocks by deleting components of their input that are not useful for forecasting.

After that, the network uses basis layers to translate the expansion coefficients $\theta_l^f$ and $\theta_l^b$ to outputs, $\hat{y}_l = g_l^f(\theta_l^f)$ and $\hat{x}_l = g_l^b(\theta_l^b)$. The operation can be described by

the following,

$$\begin{aligned} \hat{\boldsymbol{y}}_l &= \sum_{i=1}^{\mathsf{dim}(\theta_l^f)} \theta_{l,i}^f \boldsymbol{v}_i^f, \\ \hat{\boldsymbol{x}}_l &= \sum_{i=1}^{\mathsf{dim}(\theta_l^b)} \theta_{l,i}^b \boldsymbol{v}_i^b, \end{aligned} \tag{3.7}$$

where $\boldsymbol{v}_i^f$ and $\boldsymbol{v}_i^f$ are forecast and backcast basis vectors, respectively, with $\theta_{l,i}^f$ is the element of $\theta_l^f$. The function of $g_l^b$ and $g_l^f$ is to produce sufficient rich sets $\{\boldsymbol{v}_i^f\}_{i=1}^{\mathsf{dim}(\theta_l^f)}$ and $\{\boldsymbol{v}_i^b\}_{i=1}^{\mathsf{dim}(\theta_l^b)}$. The outputs of these sets can be represented by different expansion coefficients $\theta_l^f$ and $\theta_l^b$. As demonstrated below, $g_l^b$ and $g_l^f$ can be adjusted to certain functional forms to represent specific problem-specific inductive biases in order to suitably limit the structure of outputs.

NBEATS employs doubly residual stacking, which consists of two residual branches, one running over the backcast prediction of each layer and the other running over the forecast branch of each layer, that can be written as the following.

$$\boldsymbol{x}_l = \boldsymbol{x}_{l-1} - \hat{\boldsymbol{x}}_{l-1}, \quad \hat{\boldsymbol{y}}_l = \sum_l \hat{\boldsymbol{y}}_l. \tag{3.8}$$

As previously stated, the model level input $\boldsymbol{x}, \boldsymbol{x_1} \equiv \boldsymbol{x}$ is the input of the first block in the particular instance. For all other blocks, consider the backcast residual branch $\boldsymbol{x}_l$ to be a sequential examination of the input signal. The previous block eliminates the component of the signal $\hat{\boldsymbol{x}}_{l-1}$ that it can reasonably approximate, making the forecasting work on subsequent blocks easier. Moreover, this structure makes it easier to perform gradient backpropagation. Each block produces a partial prediction $\hat{\boldsymbol{y}}_l$, which is then decomposed hierarchically by being aggregated at the stack level, followed by the global network level. Therefore, the overall global output is just the sum of the partial outputs of each block (Oreshkin et al., 2019).

## 3.3 Temporal Convolutional Network

TCN was first introduced by Bai et al. (2018) by conducting systematic evaluation of generic convolutional and recurrent architectures for sequence modeling. The TCN architecture is based on recent convolutional architectures for sequential data, but is designed to combine simplicity, autoregressive prediction, and very long memory. TCNs have unique features that distinguish them from other models. Firstly, the convolutions in the architecture are causal, meaning that information from the future is not used in the prediction process. Secondly, the architecture can accommodate input sequences of any length and produce output sequences of the same length,

similar to an RNN. In addition, Bai et al. (2018) highlighted the importance of constructing highly extended effective history sizes, which refer to the ability of neural networks to analyze data from a distant past to produce predictions. This is achieved through a combination of deep networks, which are reinforced with residual layers, and dilated convolutions. For example, the TCN algorithm is much simpler than WaveNet (Oord et al., 2016) (no skip connections across layers, conditioning, context stacking, or gated activations). Compared to the language modeling architecture of Dauphin et al. (2017), TCNs do not use gating mechanisms and have much longer memory.

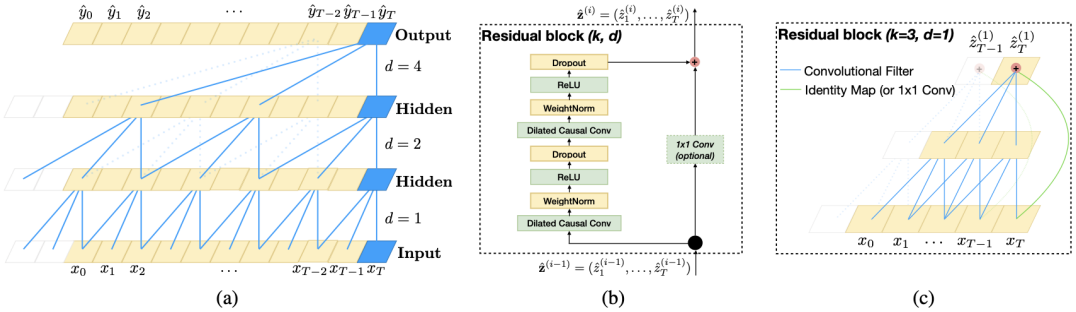


Figure 3.3: The Architecture of TCN

Suppose we have an input sequence $x_0, \ldots, x_T$, and want to predict some corresponding outputs $y_o, \ldots, y_T$ at each time. The fundamental constraint for predicting the output $y_t$ at a given time $t$ is that we can only utilize the inputs that have been observed prior to that time: $x_0, \ldots, x_t$. In formal terms, a sequence modeling network can be defined as a function $f : \mathcal{X}^{T+1} \to \mathcal{Y}^{T+1}$ that produces mapping as follows.

$$\hat{y}_0, \ldots, \hat{y}_T = f(x_0, \ldots, x_T) \tag{3.9}$$

In the setting of sequence modeling, a network $f$ is considered to be any function that maps input sequences $X = (x_0, x_1, \ldots, x_T)$ to output sequences $Y = (y_0, y_1, \ldots, y_T)$, provided it satisfies the causal constraint where the output $y_t$ only depends on the inputs $x_0, x_1, ..., x_t$ and not on any inputs in the future, $x_{t+1}, x_{t+2}, \ldots, x_T$. The objective of learning in this context is to find the network $f$ that minimizes the expected loss between the predicted outputs and the actual outputs, given by $L(y_0, y_1, \ldots, y_T, f(x_0, x_1, \ldots, x_T))$, where the input and output sequences are drawn from some distribution.

The formal definition presented here can be applied to various settings, including auto-regressive prediction where the objective is to predict a signal based on its past. In such a scenario, the target output is simply the input shifted by one time step. However, this definition does not apply to sequence-to-sequence prediction or machine translation, as these domains allow the use of the entire input sequence to predict each output (Bai et al., 2018).

TCN works by using a 1D fully-convolutional network (FCN) architecture, where each hidden layer is the same length as the input layer, and zero padding of length (kernel

size − 1) is added to keep subsequent layers the same length as previous ones. The TCN uses causal convolutions, which are convolutions where an output at time $t$ is convolved only with elements from time $t$ and earlier in the previous layer. This ensures that there is no information "leakage" from future to past. The architecture can take a sequence of any length and map it to an output sequence of the same length, just as with an RNN. To build very long effective history sizes, TCNs use a combination of very deep networks (augmented with residual layers) and dilated convolutions. This could be simply expressed by **TCN = 1D FCN + Causal convolutions**.

However, the main problem of the basic design for achieving a lengthy effective history size is that it requires an incredibly deep network or extremely large filters, neither of which were especially viable when the approaches were initially proposed. This makes it challenging to apply the aforementioned causal convolution on sequence tasks, especially those requiring longer history. To address this issue, techniques from modern convolutional architectures can be integrated into a TCN to allow for both very deep networks and very long effective history (Bai et al., 2018).

Dilated convolutions are employed in the TCN architecture to enable an extensive receptive field by skipping input values with a specific step size known as the dilation rate. This convolutional operation is used to simplify the use of causal convolution in sequence tasks that need a longer history, allowing the network to look back at an exponentially larger history corresponding to the network's depth. Therefore, dilated convolutions permit an exponentially large receptive field, which helps in applying causal convolution on sequence tasks requiring longer history. More formally, for 1-D sequence input $\boldsymbol{x} \in \mathbb{R}^n$ and a filter $f : 0, \ldots, k-1 \to \mathbb{R}$, the dilated convolution operation $F$ on element $s$ of the sequence can be written as the following

$$F(s) = (\boldsymbol{x} *_d f)(s) = \sum_{i=0}^{k-1} f(i) \cdot \boldsymbol{x}_{s-d \cdot i}, \tag{3.10}$$

where $d$ is the dilation factor, $k$ is the filter size, and $s - d \cdot i$ accounts for the direction of the past. Dilated convolutions introduce a fixed step between adjacent filter taps, resulting in a wider range of inputs being represented by the output at the top level. When the dilation rate is set to 1, the dilated convolution behaves like a regular convolution. Increasing the dilation rate allows the receptive field of a convolutional network to expand, thereby increasing its effectiveness.

There are two ways to expand the receptive field of the TCN. The first way is to increase the filter size $k$, and the second way is to increase the dilation factor $d$. In particular, the effective history of one layer is $(k-1)d$. To achieve an extremely large effective history using deep networks, it is common to increase d exponentially with the depth of the network (i.e., $d = O(2^i)$ at level $i$ of the network). This ensures that there is a filter that hits each input within the effective history while allowing for a vast deep network of effective history as illustrated in Figure 3.3 (a).

According to He et al. (2016), a residual block is described as consisting of a branch

that leads to a sequence of transformations $\mathcal{F}$. The outputs of these transformations are then added to the input $\boldsymbol{x}$ of the block as follows.

$$o = \text{Activation}(\boldsymbol{x} + \mathcal{F}(\boldsymbol{x})) \tag{3.11}$$

The use of residual blocks, as proposed by He et al. (2016), enables layers to learn modifications to the identity mapping, rather than the entire transformation, resulting in improved performance in very deep networks. However, since the receptive field of a TCN depends on its depth, filter size, and dilation factor, stabilization of larger and deeper networks becomes crucial. In situations where a large receptive field is required, such as for a history of size $2^{12}$ and a high-dimensional input sequence, a network with up to 12 layers may be necessary. Each layer is composed of multiple filters for feature extraction. Therefore, we use a generic residual module in place of a convolutional layer when designing the TCN model.

The residual block used in our baseline TCN model consists of two layers of dilated causal convolution with a ReLU non-linearity. To normalize the convolutional filters, Bai et al. (2018) applied weight normalization and added spatial dropout after each dilated convolution for regularization. Unlike standard ResNet, where the input is directly added to the output of the residual function, in TCN and ConvNets in general, the input and output can have different widths. To handle the difference in input-output widths, an additional 1x1 convolution is applied to ensure that the element-wise addition operation receives tensors of the same shape which is illustrated in Figure 3.3 (b,c).

In an experimental experiment that was conducted by Bai et al. (2018), TCN models have demonstrated superior performance when compared to generic recurrent architectures, such as LSTMs and GRUs. However, before the introduction of architectural elements like dilated convolutions and residual connections, convolutional architectures were generally weaker. The results suggest that, with these added elements, a simple convolutional architecture can outperform recurrent architectures like LSTM in various sequence modeling tasks.

## 3.4 Transformer Neural Network

Vaswani et al. (2017) introduced a novel architecture called Transformer that is built entirely on attention mechanisms, without relying on recurrence or convolutions. The proposed model architecture abandons the use of recurrence and solely relies on attention mechanisms to establish global dependencies between the input and output. Moreover, the Transformer is the first transduction model to calculate representations of its input and output using just self-attention rather than a sequence-aligned RNN or convolution.

The main working power of Transformers is from its multi-head self-attention mechanism, which has a remarkable capability of extracting semantic correlations among

elements in a long sequence. The Transformer model consists of an encoder and a decoder structure (Vaswani et al., 2017). The encoder takes an input sequence and generates a set of hidden representations for each element in the sequence. The decoder then takes these hidden representations as input and generates an output sequence. The self-attention mechanism allows the model to attend to different parts of the input sequence at different times, allowing it to capture long-term dependencies between elements.

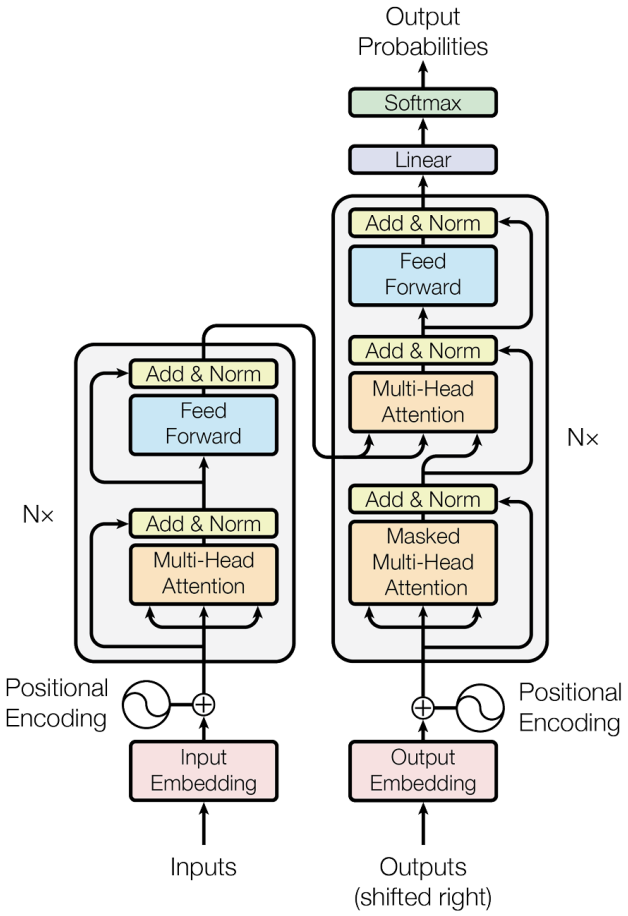


Figure 3.4: The Architecture of Transformer

To facilitate preserving some ordering information, positional encoding techniques are used to add unique position embeddings to each element in the sequence before feeding them into the Transformer. However, it is still inevitable to have temporal information loss after applying self-attention on top of them. This is usually not a serious concern for semantic-rich applications but may be problematic for time series modeling tasks where temporal ordering plays a crucial role (Vaswani et al., 2017).

The model architecture has an encoder-decoder structure, where the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\boldsymbol{z} = (z_1, \ldots, z_n)$. Given $z$, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ of one element at a time. At each step the model is auto-regressive (Graves & Graves, 2012), consuming the previously generated symbols as

additional input when generating the next.

To facilitate the encoder and decoder stacks in a Transformer model, each stack consists of multiple layers of self-attention and feed-forward neural network modules (Vaswani et al., 2017; Zeng et al., 2022). The Transformer model uses an encoder stack to process input sequences and generate hidden representations for each element in the sequence. This stack consists of a set of $N = 6$ identical layers, each of which has two sub-layers: a multi-head self-attention mechanism and a fully connected feed-forward network. Each sub-layer is followed by normalization and produces an output of dimension $d_{\text{model}} = 512$, which is added to the input of the sub-layer to create a residual connection. The use of residual connections allows the model to attend to different parts of the sequence at different times, and to generate hidden representations that can capture global dependencies between input and output.

The decoder stack receives the hidden representations generated by the encoder stack and produces the output sequence. Each layer in the decoder stack attends to both the input sequence and the output generated by previous layers, enabling it to generate predictions that depend on both the input and previous predictions. The decoder also consists of a stack of $N = 6$ identical layers, with three sub-layers in each layer. In addition to the multi-head self-attention mechanism and the fully connected feed-forward network, the decoder includes a third sub-layer that performs multi-head attention over the output of the encoder stack. Similar to the encoder, the decoder uses residual connections around each sub-layer, followed by layer normalization. The self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position $i$ depend only on the known outputs at positions less than $i$ (Vaswani et al., 2017).

To facilitate communication between the encoder and decoder stacks, an attention mechanism is used to allow the decoder to attend to different parts of the encoded input sequence at different times. This allows it to generate predictions that are informed by relevant information from the input (Wen et al., 2022). The attention function can be defined as a function that maps a query and a set of key-value pairs to an output. All of these, including the query, keys, values, and output are represented as vectors. The output is determined by a weighted sum of the values, where the weight assigned to each value is calculated using a compatibility function of the query with the corresponding key.

There are two types of attention mechanisms used in this context, namely Scaled Dot-Product Attention and Multi-Head Attention. Scaled Dot-Product Attention takes inputs in the form of queries and keys with a dimension of $d_k$, and values with a dimension of $d_v$. The dot product of the query with each key is computed, followed by division by $\sqrt{d_k}$ and applying a softmax function to derive the weights assigned to each value. In practice, the attention function is computed for a set of queries simultaneously by packing them together into a matrix $Q$ (Vaswani et al., 2017). The corresponding keys and values are also packed together into matrices $K$ and $V$, which can be expressed as follows.

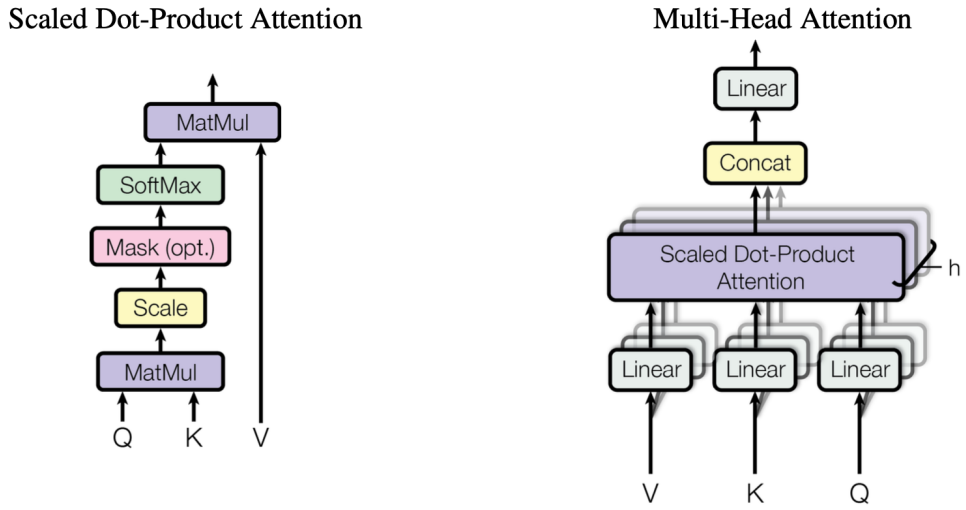


Figure 3.5: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right) \tag{3.12}$$

where queries $Q \in \mathbb{R}^{N \times d_k}$, keys $K \in \mathbb{R}^{M \times d_k}$, values $V \in \mathbb{R}^{M \times d_v}$, $N$, $M$ denote the lengths of queries and keys (values), and $d_k$, $d_v$ denote the dimensions of keys (queries) and values (Wen et al., 2022).

The multi-head attention mechanism allows the model to attend to input from multiple representational subspaces simultaneously and across different positions. It involves projecting the queries, keys, and values into several subspaces and applying the attention function independently on each of them in parallel. The output values of each subspace are concatenated and projected again to obtain the final output values, as depicted in Figure 3.5. Zeng et al. (2022) mentioned that transformer uses multi-head attention with $H$ different sets of learned projections instead of a single attention function as

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ \text{where} \quad \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{3.13}$$

where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

## 3.5 Evaluation Metrics

It is important to evaluate forecast accuracy by looking at how well the model performs on data that was not used during development. This measure provides an indication

of the magnitude of realistic forecasting mistakes. As a result, the number of residuals is a reliable indicator of the extent of realistic forecast errors. Forecast accuracy can only be determined by examining how well a model performs on fresh data that was not used during the model's development (Hyndman & Athanasopoulos, 2018).

When selecting models, it is common practice to divide the available data into two parts: training and test data. The training data helps estimate the parameters of a forecasting method, while the test data is used to evaluate its accuracy. Since the test data isn't used for predictions, it should deliver an accurate indication of how well the model will perform on new datasets.

The forecast residual is the difference between the actual value and its forecast, where residual or error refers to unpredictable part in an observation. It can be written as

$$\hat{e}_t = \hat{y}_t - y_t \tag{3.14}$$

A wide variety of forecasting performance measures is available and on a high level they can be classified into four main types: absolute, percentage, relative and scaled (Hyndman & Koehler, 2006). It is important to choose appropriate performance measures based on the data set. Absolute measures are scale-dependent, while percentage, relative, and scaled measures are scale-independent. A benefit of scale-independent measures is that they are easier to compare across different forecasts. This is especially helpful when there is a difference in the orders of magnitude of different forecasts, for example when one item is sold much more often than another.

When selecting performance measures, it is important to consider scale-independent metrics. These measurements are not dependent on the size of the data set and can be easily compared across different forecasts. This makes them particularly useful when there is a large difference in magnitude between different predictions, for example when one item sold more often than another. In this project, we use statistical error metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Symmetric Mean Absolute Percentage Error (SMAPE), Weighted Average Percentage Error (WAPE).

### a. Root Mean Squared Error (RMSE)

RMSE is a quadratic scoring mechanism that calculates the average magnitude of the error. It is also known as the metric that evaluates the quality of a forecasting model or predictor. RMSE additionally takes into account variance (the difference between anticipated values) and bias (the distance of predicted value from its true value).

$$\text{RMSE} = \sqrt{\left(\frac{1}{n}\sum_{t=1}^{n}(\hat{y}_t - y_t)^2\right)} \tag{3.15}$$

RMSE can also be compared to MAE to determine whether the forecast contains large but infrequent error. A forecast approach that minimizes RMSE generates mean

predictions, whereas one that minimizes MAE delivers median forecasts (Hyndman & Athanasopoulos, 2018).

The squared value of the residuals is used in RMSE, which magnifies the impact of outliers. The RMSE is the most significant measure in use scenarios where a few large mispredictions can be highly expensive. Moreover, the impact of squared error before averaging, RMSE gives a comparatively high weight to large errors. RMSE is more sensitive to outliers and penalises large errors more than MAE due to the fact that errors are squared initially. This means RMSE is most beneficial when large errors are very undesirable. The variation of the frequency distribution of error magnitudes increases the variance of RMSE, not the variance of the mistakes. In consequence, while being harder to comprehend, RMSE is commonly utilized.

### b. Mean Absolute Error (MAE)

MAE represents the average magnitude of the absolute values between the forecasted values and the corresponding observed values that can be written as

$$\text{MAE} = \frac{1}{n}\sum_{t=1}^{n}|\hat{y}_t - y_t| \tag{3.16}$$

where $\hat{y}$ is the forecasted value and $y$ is the actual value, and $n$ is total number of values in the test set.

MAE indicates how much inaccuracy we may expect from the forecast on an average (Armstrong, 2010). The lower the MAE number, the better the model; a value of 0 implies that the forecast is error-free. In other words, when comparing models, the model with the lowest MAE is deemed superior. If the absolute value is not used (the signs of the errors are not eliminated), the average error is known as the Mean Bias Error (MBE) and is typically used to assess average model bias. Hyndman & Koehler (2006) mentioned that MBE can convey important information but should be evaluated with caution because positive and negative mistakes would balance out.

However, MAE does not identify the proportional scale of the error, making it impossible to distinguish between large and small errors. It may be used in association with other metrics such as RMSE to assess if the errors are greater. Furthermore, MAE might obscure issues related to low data volume. MAE and RMSE can be used jointly to diagnose the variety in forecast errors. RMSE will always be more than or equal to MAE; the greater the difference between them, the greater the variation of individual residual in the sample. If RMSE equals MAE, all errors have the same magnitude.

### c. Symmetric Mean Absolute Percentage Error (SMAPE)

SMAPE was proposed by Armstrong (2010) during M3 forecasting competition that can be defined by

$$\text{SMAPE} = \frac{1}{n}\sum_{t=1}^{n}\frac{|\hat{y}_t - y_t|}{(\hat{y} + y_t)/2} \tag{3.17}$$

If $y_t$ is close to zero, $\hat{y}$ is also likely to be close to zero. Thus, the computation remains unstable since the measure still includes division by a value close to zero. Furthermore, because the value of SMAPE might be negative, the interpretation of "absolute error" can be misleading. SMAPE has a drawback in that if the actual value or forecast value is 0, the error value approaches 100%. The lower a forecast's SMAPE value, the more accurate it is.

### d. Weighted Absolute Percentage Error (WAPE)

WAPE, also known as Mean Absolute Deviation (MAD), quantifies the overall deviation between the forecasted values and the observed values. WAPE is calculated by summing the total of the observed and forecasted values and then computing the difference between these two quantities. A lower WAPE score indicates a higher level of accuracy in the model's predictions.

When the total of observed values for all time points and all items in a given backtest window is close to zero, the weighted absolute percentage error expression is undefined. In these cases, forecast produces the unweighted absolute error total, which is the numerator in the WAPE expression.

$$\text{WAPE} = \frac{\sum_{t=1}^{n}|\hat{y}_t - y_t|}{\sum_{t=1}^{n}|y_t|} \tag{3.18}$$

WAPE is less robust to outliers than RMSE since it use the absolute error rather than the squared error. Kolassa (2016), on the other hand, highlighted that one key drawback with WAPE, particularly in an intermittent demand forecasting setting, is that WAPE will be minimized in expectation in future distribution. In the case of intermittent data, this can quickly lead to zero, and the "best" forecast, in terms of MAD, may be a flat zero line.

## 3.6 Hyperparameter Optimization

This section discusses the optimization techniques employed for tuning the hyperparameters of different forecasting methods. It highlights the necessity of fine-tuning hyperparameters, as explained in Section 2.5, to attain optimal performance in the respective techniques. The problem of identifying a good value for hyperparameters $\lambda$ is called the problem of hyperparameter optimization that can be represented generally in the equation 3.19.

$$\lambda^* = \underset{\lambda\in\Lambda}{\text{argmin}}\,\mathbb{E}_{x\sim\mathcal{G}_x}[\mathcal{L}\left(x; \mathcal{A}_\lambda(X^{\text{train}})\right)] \tag{3.19}$$

where $\mathcal{G}_x$ is the unknown natural distribution of the independent observation values $\mathcal{A}_\lambda$ Bergstra & Bengio (2012). Regarding the expectation over $\mathcal{G}_x$, we utilize the commonly adopted approach of employing cross-validation explained in Section 2.6 to estimate it. By using cross-validation, we replace the expectation with a mean over a validation set $X^{(\text{valid})}$ whose elements are drawns i.i.d $x \sim \mathcal{G}_x$ which cross-validation is unbiased as long as $X^{(\text{valid})}$ is independent. Therefore, the hyperparameter optimization problem, represented by the function $\Psi$, can be formulated using Equation 3.20.

$$\begin{aligned}
\lambda^* &\approx \underset{\lambda \in \Lambda}{\text{argmin}} \underset{x \in \mathcal{X}^{(\text{valid})}}{\mu} \mathcal{L}\left(x; \mathcal{A}_\lambda(X^{\text{train}})\right) \\
&\equiv \underset{\lambda \in \Lambda}{\text{argmin}}\, \Psi(\lambda) \\
&\approx \underset{\lambda \in \{\lambda^{(1)}, \ldots, \lambda^{(S)}\}}{\text{argmin}} \Psi(\lambda) \\
&\equiv \hat{\lambda}
\end{aligned} \tag{3.20}$$

Grid search is a widely-used approach that evaluates a set of hyperparameters exhaustively and selects the best one. In grid search, the set of trials is formed by assembling every possible combination values of trial points $\lambda^{(1)}, \ldots, \lambda^{(S)}$ to find a good $\lambda$. It involves specifying a grid of hyperparameter values to search over, and evaluating the model's performance for each combination of hyperparameters.

Grid search computation is straightforward to implement and can be easily parallelized. However, Bergstra & Bengio (2012) mentioned that grid search has certain limitations. This method is most effective in low-dimensional search spaces, such as one or two dimensions, and may not perform well in high-dimensional spaces. Grid search is suitable for simple cases with a small number of hyperparameters and when working with a single time series. Therefore, in this research, grid search techniques are not utilized due to the nature of DL, which involves a large number of hyperparameters and the handling of multiple time series.

Additionally, there are advanced hyperparameter optimization frameworks to automate the process of finding an optimal hyperparameter configuration in a fast and efficient manner like Optuna and Ray Tune. Optuna offers interpretability through visualizations and can handle global models with multiple time series, while Ray Tune has automatic pruning capabilities. Akiba et al. (2019) introduced Optuna as a solution to handle the intricate nature of hyperparameters in deep learning models across diverse scenarios, encompassing both large-scale and small-scale data experiments.

Optuna is a framework for hyperparameter optimization that uses a combination of Bayesian optimization and pruning to efficiently search the hyperparameter space. Bayesian optimization uses a probabilistic model to select the next set of hyperparameters to evaluate, based on the results of previous evaluations. Optuna gradually constructs the objective function by interacting with the trial object. The search spaces are dynamically constructed during the runtime of the objective function using the methods provided by the trial object. Pruning is used to stop the evaluation of

unpromising hyperparameter combinations early, to reduce computational cost (Akiba et al., 2019).

Liaw et al. (2018) proposed Ray Tune as an alternative framework for hyperparameter optimization that incorporates a combination of grid search, random search, and adaptive search techniques to effectively explore the hyperparameter space. It includes the capability of automatic pruning to terminate the evaluation of unpromising hyperparameter combinations early on, and supports distributed computing to expedite the search process. The interface of Ray Tune fulfills the requirements for a wide range of hyperparameter search algorithms, enabling easy scalability of the search to large clusters and simplifying the implementation of algorithms.

Similar with Optuna which is an automatic software framework for hyperparameter optimization, specifically designed for DL techniques. It emphasizes an imperative and define-by-run style user API, allowing users to dynamically construct search spaces for hyperparameters. Optuna falls under the categories of "derivative-free optimization" and "black-box optimization" (Liaw et al., 2018).

# Chapter 4

# Real Dataset Results

In this chapter, we present the study results in the real-world dataset that uses historical sales data at Henkel. We compare four different DNN algorithms namely LSTM, NBEATS, TCN, and Transformer. In each section, we describe the architecture of their respective DNN. Additionally, we compare the performance of these DNN models with the current forecasting method employed by Henkel, which serves as a baseline for our analysis.

## 4.1 Data Description

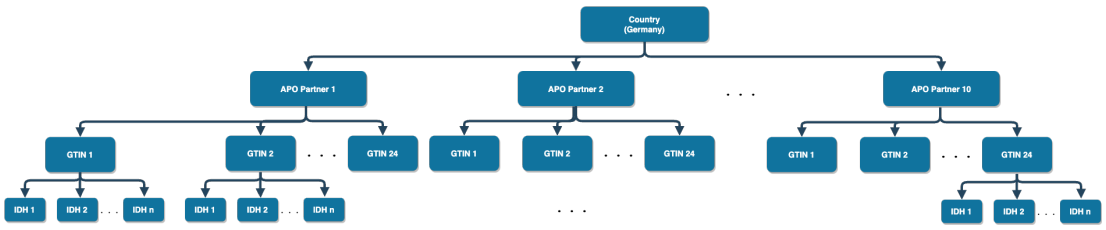


Figure 4.1: Data Structure

The dataset used in this research comprises weekly sales data of Henkel Laundry and Home Care products in Germany, spanning from January 1, 2020, to February 28, 2023. The Laundry and Home Care business consists of detergents, fabric softeners, laundry performance enhancers, automatic dishwashing, and cleaning products. More than 100 brands are under Laundry and Home Care category, with well-known products are Persil, Bref, Prill, Fleuril, and Witte Reus. The dataset is structured using a three-dimensional data model, which facilitates effective analysis and decision-making at the store level, considering specific products. Henkel operates a total of 24 factories dedicated to manufacturing Laundry and Home Care products. However, for this study, our focus is specifically on the sales areas within Germany that demonstrate the highest production volume of these products. Figure 4.1 depicted that the first dimension represents the sales area, focusing solely on the German market.

This implies that the products were manufactured in Germany and distributed within the country. Production from each factory is transported to the distribution center and subsequently to the retailers who act as customers (APO Partner) as the second dimension in Figure 4.1. The term "APO Partner" represents a group of customers located in a particular region and is associated with advanced planning and optimization strategies. It includes store location services and delivery information for customers such as LIDL, ALDI, and other supermarkets operating in a specific area. The Global Trade Item Number (GTIN) is the parent identifier for items sharing the same product name. Lastly, the International Data Harmonization (IDH) represents the specific unit of an item. For example, different sizes of Persil Power Bar, such as 472 grams and 975 grams, are categorized under a single GTIN, where Persil represents the GTIN and IDH denotes the specific variant of Persil based on size.

In this research, we aggregate the IDH level into their respective parent IDs known as GTIN. This aggregation allows us to establish time series granularity based on the APO Partner and GTIN levels. As a result, we grouped those hierarchical structures into one global model. Due to the computational limitation, we restrict our research to 24 APO Partners and 10 GTIN, resulting in a total of 240-time series for analysis.

Table 4.1: Data Description

| **Features** | **Description** |
|---|---|
| Actual Sales | The historical sales (target value) |
| IDH | International Data Harmonization that means item or product |
| APO Partner | Customers to whom we will deliver the products that means stores or retailers who ordered Henkel products |
| GTIN | Parent ID that is one category above IDH that means the brand of all sizes for each specific product |
| Date | The data in "y-m-d" format |
| Day | The day based on date |
| Month | The month of the date |
| Year | The year of the date |
| Week | The week of the year |

Table 4.1 provides a detailed description of the data attributes utilized in this research. The primary data attribute is the actual sales data, which constitutes the historical time series. Additionally, we incorporate time calendar variations such as day, month, year, and week number. The granularity of the forecasting is specified on a weekly basis, considering the GTIN and APO Partner within the German region. To ensure the integrity of the analysis, the data is partitioned into three sets: the training set, utilized for model fitting; the validation set, employed to assess the model's performance; and the test set, employed to evaluate the model's ability to handle unforeseen data. Since historical sales data is a sequence based on time series, randomizing the cross-validation technique is not feasible. Because the prediction

objective in time series analysis involves predicting future values, the test data must have a higher index than the training set. This ensures that the test data always represents future time points compared to the data used for model training. In our study, the model fitting process encompasses the period from 1st January 2020 to 15th April 2022, while the validation data spans from 16th April 2022 to the end of December 2022. The test set, on the other hand, covers the period from 1st January 2023 to the end of February 2023, enabling a comparison with the actual sales data recorded until February 2023 that is 7 weeks.

## 4.2 LSTM Performance

During prediction, the LSTM model incorporates the previous target value, previous hidden state, and covariates at time $t$ to forecast the target at time $t$. To achieve the desired performance, this research utilized particular hyperparameters, as described in Section 2.5. The first hyperparameter set is the **input chunk length**, initially set to 14, implying that the model will take the previous 14 time steps as input to make predictions. Moreover, **training length** refers to the length of the time series used during training, including both input (target and covariates) and output (target) time series. The length of the training time series should be set to a higher value than the input chunk length because the RNN should run for as many iterations as it will during inference. We set the training length to 30, which means the model will be trained using a time series of length 30, including the past 14 time steps as inputs and the next time step as output. The model will be trained to predict the target values at time steps 15 through 30 based on the past 14 time steps and any available covariate data.

The **hidden dimension** or hidden size refers to the number of hidden units or neurons in the LSTM layer, which we set initially to 7. These hidden units process the input sequence and create a new representation that is passed to the output layer. The higher the hidden dimension, the more complex the model can be and the better it can capture complex patterns in the input data. However, a higher hidden dimension also increases the number of model parameters, which can lead to overfitting and longer training times. Besides that, the **number of RNN layers** needs to be considered based on the complexity of the problem and the available computational resources. Due to the limitation of computational resources in this study, we create an initial LSTM model with a single stacked layer.

**Batch size** is a hyperparameter representing the number of time series (input and output sequences) used in each training pass. Setting a larger batch size can improve training efficiency by allowing the model to process more samples simultaneously, while setting a smaller batch size can help prevent overfitting and improve generalization. However, larger batch sizes may require more memory, resulting in slower convergence. The initial batch size is set to 32, which is the default option.

The **learning rate** initially set to 0.001 determines the degree of adjustment to

the neural network weights based on the loss gradient during backpropagation. The subsequent hyperparameter to be considered is an **epoch**, which signifies a complete iteration of the neural network on the entire training dataset. In this study, the initial epoch value is set to 100. Another crucial hyperparameter is the **random state**, which is used to manage the random weight initialization. This hyperparameter is discussed in detail in Sub-section 2.5.2 and is set to 10.

Those initial hyperparameter settings mentioned earlier may not be optimal for the model's performance. Therefore, experimentation is necessary to find the optimal values that can be found through a combination of cross-validation and optimization techniques, which are explained in Section 3.6.

Table 4.2: Hyperparameter Setting for LSTM

| Hyperparameter | Initial Values | Optimal Values |
|---|---|---|
| Input Chunk Length | 14 | 14 |
| Training Length | 30 | 30 |
| Hidden Dimension | 7 | 7 |
| Number of RNN Layers | 1 | 3 |
| Learning Rate | 0.01 | 0.001 |
| Batch Size | 16 | 32 |
| Epoch | 100 | 300 |

Through the implementation of a 5-fold cross-validation and an optimization approach, a set of hyperparameters was evaluated with various values assigned to each hyperparameter. Table 4.2 presented the findings that setting up the number of RNN layers to 3 was better than using a single layer. A higher number of layers can help capture more complex patterns in the data. However, it also increases the number of parameters in the model, which may require more training time and computational resources. Additionally, the optimal batch size was determined to be 16 through various experiment setups to avoid overfitting and improve the generalization of the model. This research shows that the model will produce the optimal result with a learning rate = 0.001. Thus, it ensures that the same initial weights are used for each training run, making the model result reproducible.

The optimal hyperparameter settings were determined by evaluating the model's performance based on the evaluation error and computation time, which are detailed in the table below.

Table 4.3: LSTM Performance through Hyperparameter Setting

| Performance | Initial Hyperparameter | Optimal Hyperparameter |
|---|---|---|
| RMSE | 5818.29 | 5402.12 |
| Computation Time | 01:06:39.04 | 01:45:28.01 |

The process of optimizing hyperparameters involved calculating the RMSE in the validation data, and a lower value is desirable. Table 4.3 depicted that in the optimal hyperparameter setup, the RMSE was found to be smaller than that in the initial setup. Nevertheless, the optimal model's computational time is longer than the initial model since the epoch value is also higher. Before determining the optimal values, multiple values were assigned to each hyperparameter to explore their effects. The initial hyperparameters were carefully determined based on the characteristics of the real data, adjusting the definitions of each hyperparameter as discussed in Section 2.5. Consequently, the optimal hyperparameters closely align with the initially selected values.

## 4.3 NBEATS Performance

The next DNN algorithm utilized in this research is NBEATS, which is a novel approach that can handle multivariate series and covariates by flattening the inputs of the model to a 1-D series and then reshaping the outputs to a tensor of appropriate dimensions. To determine the length of the input sequence that is fed into the model, the hyperparameter **input chunk length** is used, which was initially set to 7, along with the **output chunk length** that was also set to 7. The output chunk length is crucial in determining the length of the forecast model, as the aim is to predict at least 7 weeks ahead.

As described in Section 3.2, there are two types of NBEATS architecture: the generic and interpretable architecture consisting of one trend and one seasonality stack with appropriate waveform generator functions. The hyperparameter **generic architecture** determines whether the generic architecture of NBEATS is used or not. In this study, we set the generic architecture to be true because we are not aiming to achieve interpretable results using this model. Since we have set the generic architecture to True, the **number of stacks** represents the number of stacks to comprise the entire model, which initially set to 10. It is essential to avoid overfitting the model to the training data, such that it is recommended to employ techniques such as early stopping and regularization to prevent the model from becoming too complex.

The **hidden dimension** is an important parameter in the NBEATS model as it affects the size of the expansion coefficients for each layer. The initial value for this parameter is set to 3 dimensions. Moreover, the default value of **batch size** is 32, and we use this value as our starting point. The initial **number of blocks** used in deployed NBEATS is 1, which is necessary to build blocks stacked on top of each other to create a deep neural network. In addition, the parameter for the **number of layers** is set to 4 as a starting value, which determines the quantity of fully connected layers in each of the stack layers. In order to maintain consistency with other models, the choice of activation function is a critical hyperparameter. In this study, we use the same activation function as in other models that is **ReLU**. Moreover, we set the **number of epochs** to 100 and initialize the **random state** to 423. This is the same initial value we set for RNN in our experiment.

Table 4.4: Hyperparameter Setting for NBEATS

| Hyperparameter | Initial Values | Optimal Values |
|---|---|---|
| Input Chunk Length | 30 | 30 |
| Output Chunk Length | 7 | 7 |
| Number of Stacks | 10 | 30 |
| Hidden Dimensions | 3 | 3 |
| Batch Size | 32 | 32 |
| Number of Blocks | 1 | 3 |
| Number of Layers | 4 | 4 |
| Number of Epochs | 100 | 800 |

Table 4.4 presents the hyperparameters used in the NBEATS model, along with their initial and optimal values. These hyperparameters play an essential role in determining the performance and accuracy of the NBEATS model, and their optimal values were determined through experimentation and 5-fold validation. After conducting a hyperparameter search, the optimal number of blocks was determined to be 3, while the optimal number of stacks was found to be 30, which differed from the initial value of 10. For the remaining hyperparameters, multiple values were tested before ultimately converging to the same values as the initial ones.

Table 4.5: NBEATS Performance through Hyperparameter Setting

| Performance | Initial Hyperparameter | Optimal Hyperparameter |
|---|---|---|
| RMSE | 5464.36 | 4728.09 |
| Computation Time | 00:25:06.03 | 00:32:18.08 |

Table 4.5 summarizes the performance of the NBEATS model with different hyperparameter settings. The model's performance is evaluated based on two metrics, the evaluation error by RMSE and the computation time. The table compares the performance of the model with its initial hyperparameter values and optimal hyperparameter values. The initial RMSE was 5464.36, which decreased to 5454.9 with optimal hyperparameters.

## 4.4 TCN Performance

In TCN, the **input chunk length**, which represents the number of past time steps fed to the forecasting module, is set to 30. Additionally, the **output chunk length**, which determines the length of the forecasted sequence, is set to 7. The initial dropout rate for each convolutional layer is set to 0.1 and later tested with a value of 0.2 in the subsequent search. Through optimization, it was determined that the

optimal dropout rate is 0.1. Other hyperparameters are detailed in Table 4.6 with the main difference with other DNN algorithms lies on **dilation**. Its architecture includes dilated convolutions that allow the model to have a wider receptive field and capture patterns over longer sequences. This enables TCN to capture complex temporal dependencies in the data, which is crucial for accurate forecasting. In addition, **the number of filters** refers to the number of channels or feature maps generated by the convolutional layers in the network. Each filter captures different patterns or features from the input data. By adjusting the number of filters, we can control the complexity and capacity of the TCN model. Initially, we set the number of filters to 3 and subsequently explored various values to identify the optimal configuration. Through experimentation, we determined that setting the number of filters to 5 yielded the best performance in terms of capturing and representing the underlying patterns and features in the data.

Table 4.6: Hyperparameter Setting for TCN

| Hyperparameter | Initial Values | Optimal Values |
|---|---|---|
| Input Chunk Length | 30 | 30 |
| Output Chunk Length | 7 | 7 |
| Dropout | 0.1 | 0.1 |
| Dilation | 1 | 2 |
| Number of filters | 3 | 3 |
| Kernel size | 3 | 5 |
| Batch Size | 32 | 32 |
| Number of Epochs | 100 | 500 |

Table 4.7: TCN Performance through Hyperparameter Setting

| Performance | Initial Hyperparameter | Optimal Hyperparameter |
|---|---|---|
| RMSE | 5285.05 | 4318.74 |
| Computation Time | 01:40:20.89 | 01:59:48.18 |

Through the hyperparameter optimization process of TCN, significant improvements were achieved. The error evaluation was reduced by 14%, resulting in a lower value of 4532.84. Additionally, the optimization led to improved computational efficiency, enabling faster processing time and the generation of more accurate forecasting results. Although the TCN model has a longer training computation time, it is able to capture and model complex temporal patterns and dependencies in the data more effectively. As a result, the TCN model outperforms other DNN algorithms in terms of predictive accuracy, as reflected by the lower RMSE value. This suggests that the additional training time invested in the TCN model is justified by the improved accuracy of its forecasts.

## 4.5 Transformer Performance

The initial hyperparameter configuration for the Transformer model in the Darts package is consistent with the TCN model discussed in the previous section, including the dropout rate. However, the notable difference lies in the number of **encoder layers** and **decoder layers**. The encoder layers are crucial in encoding the input time series sequence and extracting its underlying features. Increasing the number of encoder layers has the potential to enable the model to capture more intricate temporal patterns and dependencies within the data. On the other hand, the decoder layers are responsible for generating the output sequence based on the encoded input sequence. By increasing the number of decoder layers, the model's capacity to capture complex patterns and dependencies in the time series data can be improved. In the case of the Transformer model, both the encoder and decoder layers were initially set to 2, and through hyperparameter optimization, the optimal number of layers was determined to be 3.

Table 4.8: Hyperparameter Setting for Transformer

| Hyperparameter | Initial Values | Optimal Values |
|---|---|---|
| Input Chunk Length | 30 | 30 |
| Output Chunk Length | 7 | 7 |
| Dropout | 0.1 | 0.2 |
| Batch Size | 32 | 32 |
| Number of Encoder Layers | 2 | 3 |
| Number of Decoder Layers | 2 | 3 |
| Number of Epochs | 100 | 200 |

Table 4.9: Transformer Performance through Hyperparameter Setting

| Performance | Initial Hyperparameter | Optimal Hyperparameter |
|---|---|---|
| RMSE | 5530.42 | 4433.02 |
| Computation Time | 01:07:17.96 | 01:23:10.06 |

Table 4.9 highlights the significant impact of hyperparameter tuning on the performance of the Transformer model, as evidenced by the reduction in RMSE and computation time. The computation time for training the Transformer model to compute the optimal hyperparameters is comparable to that of the LSTM model in Section 4.2. Despite their different architectures, both models require a similar amount of computational resources and time to train. Furthermore, the Transformer model's evaluation error is similar to that of the LSTM model. This suggests that both models perform comparably in terms of their predictive accuracy on sales forecasting.

## 4.6 Performance Comparison Results

To demonstrate the effectiveness of the novel DNN algorithms implemented in this thesis, a comparative analysis is conducted using a real sales dataset at Henkel, as discussed in Sections 4.2 to 4.5. The evaluation begins by comparing the performance of each algorithm in terms of evaluation error on the test set, as outlined in Section 3.5. The evaluation error of the validation data, which was previously discussed in the preceding section, was employed for hyperparameter optimization purposes. However, at this stage, we aim to compare the performance on the test set of optimal hyperparameters to assess the models' ability to capture unforeseen data, which can be seen in Figure 4.10.

Table 4.10: Error Evaluation of Test Data

| Evaluation Error | LSTM | NBEATS | TCN | Transformer |
|---|---|---|---|---|
| RMSE | 4187.24 | 3692.7 | 3224.53 | 3686.15 |
| MAE | 1563.92 | 1105.46 | 1335.31 | 1448.49 |
| SMAPE | 1.67 | 1.71 | 1.66 | 1.67 |
| WAPE | 2.3 | 1.62 | 1.96 | 2.13 |

The evaluation of the test set, conducted with the optimal hyperparameters, confirms that TCN is the top-performing DNN algorithm. This outcome is consistent with the validation results, which also identified TCN as the most effective approach. Analysis of the validation data across previous sections shows that TCN consistently outperforms other models, exhibiting the lowest error metrics. According to the test data, TCN achieves the lowest error of RMSE with a value of 3224.53, followed closely by Transformer with an error of 3686.15.

In terms of MAE, NBEATS has the lowest error with a value of 1105.46, followed by the Transformer with an error of 1448.49. However, we consider the best performing model based RMSE since RMSE is more sensitive to larger errors and provides an overall measure of accuracy. As explained in Section 3.5, RMSE is more sensitive to larger errors and provides an overall measure of accuracy because it penalizes larger errors more heavily, as squaring amplifies the effect of larger errors.

Table 4.11: Error Evaluation Compared to the Current Forecast

| Evaluation Error | LSTM | NBEATS | TCN | Transformer | Current Method |
|---|---|---|---|---|---|
| RMSE | 5474.90 | 4779.91 | 4532.84 | 4890.35 | 6174.65 |
| MAE | 1323.96 | 1666.82 | 1237.44 | 1338.25 | 2326.38 |
| SMAPE | 1.42 | 1.30 | 1.26 | 1.27 | 1.56 |
| WAPE | 1.04 | 1.31 | 0.97 | 1.05 | 1.83 |

In the current forecasting method implemented in Henkel, predictions are made for a one-week time horizon, and the model is retrained by appending the most recent

week's data to predict the subsequent week. To make it comparable, we evaluate the accuracy of the one-week ahead predictions for all APO partner and GTIN combinations within the time series. It shows that TCN model performs best for sales forecasting in the SCM, as it has the smallest evaluation error.

The existing forecasting approach at Henkel employs exponential smoothing with customized adjustments made by the demand planning team, where predictions are made on a series-by-series basis. In this research, we utilize global models that can simultaneously predict all series with a specific focus on weekly forecasts for each APO Partner-GTIN combination. By incorporating calendar variations such as day, month, year, and week number, we aim to enhance the accuracy of predictions compared to univariate time series methods like exponential smoothing. Additionally, the computational time required to train the DNN algorithms is relatively fast. The 240-time series data training process required almost two hours for TCN, while NBEATS exhibited a faster training time of only a half an hour.

# Chapter 5

# Simulation Data Results

Simulation studies offer a valuable approach for comprehending the behaviour of statistical methods, as they provide insights into the performance of these methods when the underlying "truth" is known through the generation process of the data. Simulation studies enable us to gain a deeper understanding of the methods by examining properties such as bias. In this context, the term "data-generating mechanism" refers to the process by which random numbers are utilized to create an artificial dataset. The objective of this experiment is to validate the performance of the best-performing DNN model using the optimal hyperparameters obtained in the previous chapter.

## 5.1 Simulation Settings

The distribution of sales data in Henkel exhibits characteristics consistent with an exponential distribution which can be seen in Appendix C. To create the simulation data, we utilize the same parameter value as the sales data, specifically $\lambda = 1717.63$. To better understand the data, a log normal transformation is applied and normalized into range 0 to 1. This transformation aids in visualizing the data and potentially normalizing its distribution. To do this, we build time series datasets that are autocorrelated. In this experiment, we set the autocorrelation value to 0.6, indicating that the subsequent data points have a positive correlation with the preceding data points, in accordance with the characteristics of a time series function. In addition, the performance of the DNN algorithm is examined under a sample size of n=100,000 to assess its effectiveness.

Furthermore, we use the specified hyperparameters to evaluate the effectiveness of the model for the simulated data. In this study, the simulation data is divided into training and validation datasets. The optimized hyperparameters obtained from the real dataset are then applied to the simulation data. A log transformation is applied to address the right-skewness of the exponential data with the small values that occur more frequently than higher values. By transforming the data into a log-normal distribution within the range from 0 to 1, we can achieve a more balanced distribution

where smaller values occur more frequently compared to higher values.

The comparison of the loss function, particularly RMSE, on the validation simulation data serves as the basis for conducting a robustness analysis and evaluating the performance of the DNN models. By analyzing the loss values in both the train and the validation dataset, we can assess the effectiveness and reliability of the DNN algorithms in accurately predicting the simulated data.

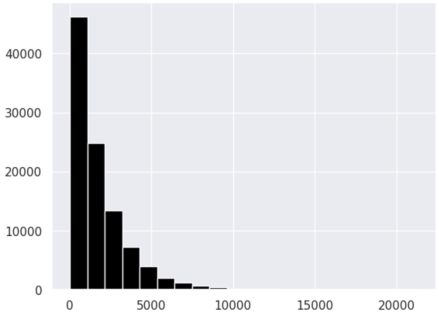


Figure 5.1: Data-generating Distribution

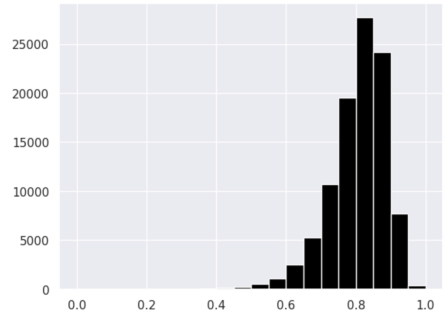


Figure 5.2: Log-normal Transformation

## 5.2 Experiment Results

Based on the performance comparison results presented in Section 4.6, it can be concluded that TCN model performs the best among the evaluated models. Additionally, NBEATS model demonstrates the fastest computation method. These findings suggest that the TCN model achieves superior performance in terms of accuracy, while the NBEATS model excels in computational efficiency. In order to assess the performance of the models, the simulated data is evaluated, taking into account both the evaluation error and computation constraints.

Table 5.1: Error Evaluation of Simulated Data

| Evaluation Error | LSTM | NBEATS | TCN | Transformer | Exponential |
|---|---|---|---|---|---|
| RMSE | 9.63 | 10.36 | 9.59 | 9.67 | 12.9 |
| MAE | 7.67 | 8.22 | 7.59 | 7.66 | 10.45 |
| SMAPE | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |
| WAPE | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |

This evaluation is conducted in all models to analyze their effectiveness in handling the simulated data. The TCN model demonstrated superior performance, as evidenced by its minimal evaluation error, as shown in Table 5.1. This outcome aligns with the findings from the analysis of the real dataset, where TCN also emerged as the top-performing model. Despite employing the optimal hyperparameters identified in the previous section. This consistency further reinforces the superiority of TCN, as

it utilizes temporal convolutions to capture complex temporal dependencies across multiple time steps. Compared to the baseline model, exponential smoothing, all four DNN models perform better. Therefore, TCN is well-suited for effectively modelling intricate patterns like seasonality and temporal changes in time series data.

The simulated data was trained using a fixed epoch value of 100 to ensure consistency and facilitate a fair comparison in determining the optimal model. Subsequently, the loss function was computed for both the training and validation datasets. The resulting loss values are visualized in Figure 5.3 for the NBEATS model and Figure 5.4 for the TCN model. These figures provide a graphical representation of how the loss function evolves during the training process for each model, highlighting the convergence and performance of the models over time.

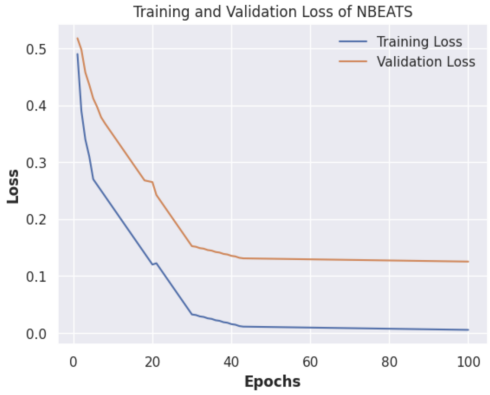


Figure 5.3: Loss Values of NBEATS

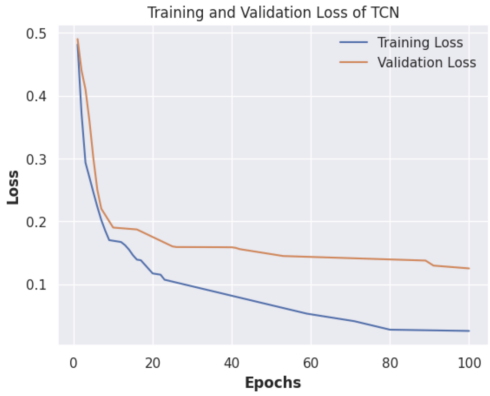


Figure 5.4: Loss Values of TCN

The training process of the TCN model requires a longer duration compared to the NBEATS model, specifically when employing the optimal hyperparameters of 3 blocks and 3 filters. Despite the increased training time, the TCN model exhibits enhanced performance in capturing the temporal dependencies and patterns within the data. This finding suggests that the additional training time invested in the TCN model is justified by its ability to effectively learn and model complex relationships in the time series data, ultimately leading to improved forecasting accuracy. TCN benefits from parallelism in computation, as convolutions can be performed concurrently across different time steps. This parallel processing capability speeds up training and inference, making TCN more efficient than sequential models like LSTM.

# Chapter 6

# Conclusion and Recommendations

## 6.1 Conclusion

The DNN models with a global forecasting approach, which involve selecting the optimal forecasting technique for each individual item, show promising potential for achieving superior performance in sales forecasting within the SCM context. Among the evaluated models, the TCN stands out as the best performing model, exhibiting the smallest evaluation error and generating forecasts that closely resemble the actual sales trends. Through the evaluation of the forecast results using the testing data, it can be concluded that the selected DNN model outperforms the current forecasting method used in Henkel. Implementation of the best model results in an average improvement of 10% in forecast accuracy compared to the current forecasting method, enabling the adoption of more sophisticated forecasting models.

Moreover, the analysis of the simulated data demonstrated that the TCN model outperforms other models in terms of accuracy, displaying consistency with the results obtained from real-world data. It is important to note that the TCN model does require additional computation time and exhibits a slower convergence towards the minimum error compared to the alternative models. This finding highlights the trade-off between computation time and forecast accuracy, indicating that the TCN model is a promising choice for achieving highly precise predictions.

## 6.2 Recommendations

In order to enhance the forecasting performance, it is worth considering the possibility of ensembling the TCN and NBEATS models in future research. This combination holds potential advantages as TCN exhibits the smallest error and NBEATS has the advantage of shorter training time. By leveraging the strengths of both models, the

ensemble approach could lead to enhanced forecasting accuracy while maintaining computational efficiency.

Furthermore, it is recommended to incorporate additional external factors in future research. Consideration should be given to factors from various domains, including weather conditions, economic indicators, events and promotions. By incorporating these external factors into the forecasting models, a more comprehensive and holistic understanding of the underlying dynamics influencing sales patterns can be achieved. This has the potential to improve the accuracy and reliability of the forecasts, thereby enhancing the decision-making processes in supply chain management.

# References


Agrawal, D., & Schorling, C. (1996). Market share forecasting: An empirical comparison of artificial neural networks and multinomial logit model. *Journal of Retailing*, *72*(4), 383–407.

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th acm sigkdd international conference on knowledge discovery & data mining* (pp. 2623–2631).

Armstrong, J. S. (2010). Long-range forecasting, 2nd. *Available at SSRN 666990*.

Baecke, P., De Baets, S., & Vanderheyden, K. (2017). Investigating the added value of integrating human judgement into statistical demand forecasting systems. *International Journal of Production Economics*, *191*, 85–96.

Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *Neural Networks: Tricks of the Trade: Second Edition*, 437–478.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157–166.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, *13*(2).

Bishop, C. M., & Nasrabadi, N. M. (2006). *Pattern recognition and machine learning* (Vol. 4) (No. 4). Springer.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of compstat'2010: 19th international conference on computational statisticsparis france, august 22-27, 2010 keynote, invited and contributed papers* (pp. 177–186).

Brockwell, P. J., & Davis, R. A. (2002). *Introduction to time series and forecasting*. Springer.

Brown, R. G., & Meyer, R. F. (1961). The fundamental theorem of exponential smoothing. *Operations Research*, *9*(5), 673–685.

Cecaj, A., Lippi, M., Mamei, M., & Zambonelli, F. (2020). Comparing deep learning and statistical methods in forecasting crowd distribution from aggregated mobile phone data. *Applied Sciences*, *10*(18), 6580.

Chen, I.-F., & Lu, C.-J. (2017). Sales forecasting by combining clustering and machine-learning techniques for computer retailing. *Neural Computing and Applications*, *28*, 2633–2647.

Chen, X.-W., & Lin, X. (2014). Big data deep learning: challenges and perspectives. *IEEE access*, *2*, 514–525.

Chen, Y., Kang, Y., Chen, Y., & Wang, Z. (2020). Probabilistic forecasting with temporal convolutional neural network. *Neurocomputing*, *399*, 491–501.

Collobert, R., & Bengio, S. (2004). Links between perceptrons, mlps and svms. In *Proceedings of the twenty-first international conference on machine learning* (p. 23).

Dauphin, Y. N., Fan, A., Auli, M., & Grangier, D. (2017). Language modeling with gated convolutional networks. In *International conference on machine learning* (pp. 933–941).

Efat, M. I. A., Hajek, P., Abedin, M. Z., Azad, R. U., Jaber, M. A., Aditya, S., & Hassan, M. K. (2022). Deep-learning model using hybrid adaptive trend estimated series for modelling and forecasting sales. *Annals of Operations Research*, 1–32.

Fildes, R., Ma, S., & Kolassa, S. (2022). Retail forecasting: Research and practice. *International Journal of Forecasting*, *38*(4), 1283–1318.

Gahirwal, M. (2013). Inter time series sales forecasting. *arXiv preprint arXiv:1303.0117*.

Gamboa, J. C. B. (2017). Deep learning for time-series analysis. *arXiv preprint arXiv:1701.01887*.

Godahewa, R., Bandara, K., Webb, G. I., Smyl, S., & Bergmeir, C. (2021). Ensembles of localised models for time series forecasting. *Knowledge-Based Systems*, *233*, 107518.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Goodfellow, I., Courville, A., & Bengio, Y. (2012). Spike-and-slab sparse coding for unsupervised feature discovery. *arXiv preprint arXiv:1201.3382*.

Goodfellow, I., Lee, H., Le, Q., Saxe, A., & Ng, A. (2009). Measuring invariances in deep networks. *Advances in neural information processing systems*, *22*.

Grandvalet, Y., & Bengio, Y. (2004). Semi-supervised learning by entropy minimization. *Advances in neural information processing systems*, *17*.

Graves, A., & Graves, A. (2012). Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, 37–45.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.* A field guide to dynamical recurrent neural networks. IEEE Press In.

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

Hosseinnia, S., Fahimeh, Ebrahimi, G., & Ali. (2022). Applications of deep learning into supply chain management: a systematic literature review and a framework for future research. *Artificial Intelligence Review*, 1–43.

Hyndman, R. J., Ahmed, R. A., Athanasopoulos, G., & Shang, H. L. (2011). Optimal combination forecasts for hierarchical time series. *Computational statistics & data analysis*, *55*(9), 2579–2589.

Hyndman, R. J., & Athanasopoulos, G. (2018). *Forecasting: principles and practice*. OTexts.

Hyndman, R. J., & Koehler, A. B. (2006). Another look at measures of forecast accuracy. *International Journal of Forecasting*, *22*(4), 679–688.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (Vol. 112). Springer.

Janiesch, C., Zschech, P., & Heinrich, K. (2021). Machine learning and deep learning. *Electronic Markets*, *31*(3), 685–695.

Jiménez, F., Sánchez, G., García, J. M., Sciavicco, G., & Miralles, L. (2017). Multi-objective evolutionary feature selection for online sales forecasting. *Neurocomputing*, *234*, 75–92.

Kharfan, M., Chan, V. W. K., & Firdolas Efendigil, T. (2021). A data-driven forecasting approach for newly launched seasonal products by leveraging machine-learning approaches. *Annals of Operations Research*, *303*(1-2), 159–174.

Kolassa, S. (2016). Evaluating predictive count data distributions in retail sales forecasting. *International Journal of Forecasting*, *32*(3), 788–803.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90.

Laptev, N., Yosinski, J., Li, L. E., & Smyl, S. (2017). Time-series extreme event forecasting with neural networks at uber. In *International conference on machine learning* (Vol. 34, pp. 1–5).

Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., & Ng, A. Y. (2011). On optimization methods for deep learning. In *Proceedings of the 28th international conference on international conference on machine learning* (pp. 265–272).

Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y.-X., & Yan, X. (2019). Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in neural information processing systems*, *32*.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., & Stoica, I. (2018). Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

Loureiro, A. L., Miguéis, V. L., & da Silva, L. F. (2018). Exploring the use of deep neural networks for sales forecasting in fashion retail. *Decision Support Systems*, *114*, 81–93.

Ma, S., & Fildes, R. (2021). Retail sales forecasting with meta-learning. *European Journal of Operational Research*, *288*(1), 111–128.

Ma, S., Fildes, R., & Huang, T. (2016). Demand forecasting with high dimensional data: The case of sku retail sales forecasting with intra-and inter-category promotional information. *European Journal of Operational Research*, *249*(1), 245–257.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018a). The m4 competition: Results, findings, conclusion and way forward. *International Journal of Forecasting*, *34*(4), 802–808.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018b). Statistical and machine learning forecasting methods: Concerns and ways forward. *PloS one*, *13*(3), e0194889.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2022a). M5 accuracy competition: Results, findings, and conclusions. *International Journal of Forecasting*, *38*(4), 1346–1364.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2022b). The m5 competition: Background, organization, and implementation. *International Journal of Forecasting*, *38*(4), 1325–1336.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2022c). Predicting/hypothesizing the findings of the m5 competition. *International Journal of Forecasting*, *38*(4), 1337–1345.

Mariet, Z., & Kuznetsov, V. (2019). Foundations of sequence-to-sequence modeling for time series. In *The 22nd international conference on artificial intelligence and statistics* (pp. 408–417).

Mesnil, G., Dauphin, Y., Glorot, X., Rifai, S., Bengio, Y., Goodfellow, I., ... others (2012). Unsupervised and transfer learning challenge: a deep learning approach. In *Proceedings of icml workshop on unsupervised and transfer learning* (pp. 97–110).

Montero-Manso, P., Athanasopoulos, G., Hyndman, R. J., & Talagala, T. S. (2020). Fforma: Feature-based forecast model averaging. *International Journal of Forecasting*, *36*(1), 86–92.

Montero-Manso, P., & Hyndman, R. J. (2021). Principles and algorithms for forecasting groups of time series: Locality and globality. *International Journal of Forecasting*, *37*(4), 1632–1653.

Montgomery, D. C., Jennings, C. L., & Kulahci, M. (2015). *Introduction to time series analysis and forecasting*. John Wiley & Sons.

Moritz, S., Sardá, A., Bartz-Beielstein, T., Zaefferer, M., & Stork, J. (2015). Comparison of different methods for univariate time series imputation in r. *arXiv preprint arXiv:1510.03924*.

Moulines, E., & Bach, F. (2011). Non-asymptotic analysis of stochastic approximation algorithms for machine learning. *Advances in neural information processing systems*, *24*.

Niu, T., Wang, J., Lu, H., Yang, W., & Du, P. (2020). Developing a deep learning framework with two-stage feature selection for multivariate financial time series forecasting. *Expert Systems with Applications*, *148*, 113237.

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., ... Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Oreshkin, B. N., Carpov, D., Chapados, N., & Bengio, Y. (2019). N-beats: Neural basis expansion analysis for interpretable time series forecasting. *arXiv preprint arXiv:1905.10437*.

Panagiotelis, A., Athanasopoulos, G., Gamakumara, P., & Hyndman, R. J. (2021). Forecast reconciliation: A geometric view with new insights on bias correction. *International Journal of Forecasting*, *37*(1), 343–359.

Paria, B., Sen, R., Ahmed, A., & Das, A. (2021). Hierarchically regularized deep forecasting. *arXiv preprint arXiv:2106.07630*.

Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y. (2013). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

Ross, S. M. (2014). *Introduction to probability models*. Academic press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Salinas, D., Flunkert, V., Gasthaus, J., & Januschowski, T. (2020). Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, *36*(3), 1181–1191.

Schervish, M. J., & DeGroot, M. H. (2012). *Probability and statistics*. Pearson Education.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, *61*, 85–117.

Smyl, S. (2020). A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. *International Journal of Forecasting*, *36*(1), 75–85.

Spiliotis, E., Makridakis, S., Semenoglou, A.-A., & Assimakopoulos, V. (2020). Comparison of statistical and machine learning methods for daily sku demand forecasting. *Operational Research*, 1–25.

Spiliotis, E., Petropoulos, F., & Assimakopoulos, V. (2019). Improving the forecasting performance of temporal hierarchies. *PLoS One*, *14*(10), e0223422.

Sun, Z.-L., Choi, T.-M., Au, K.-F., & Yu, Y. (2008). Sales forecasting using extreme learning machine with applications in fashion retailing. *Decision Support Systems*, *46*(1), 411–419.

Theodorou, E., Wang, S., Kang, Y., Spiliotis, E., Makridakis, S., & Assimakopoulos, V. (2022). Exploring the representativeness of the m5 competition data. *International Journal of Forecasting*, *38*(4), 1500–1506.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Walpole, R. E., Myers, R. H., Myers, S. L., & Ye, K. (1993). *Probability and statistics for engineers and scientists* (Vol. 5). Macmillan New York.

Weller, M., & Crone, S. F. (2012). Supply chain forecasting: Best practices & benchmarking study.

Wen, Q., Zhou, T., Zhang, C., Chen, W., Ma, Z., Yan, J., & Sun, L. (2022). Transformers in time series: A survey. *32nd International Joint Conference on Artificial Intelligence (IJCAI 2023)*.

Weng, T., Liu, W., & Xiao, J. (2020). Supply chain sales forecasting based on lightgbm and lstm combination model. *Industrial Management & Data Systems*, *120*(2), 265–279.

Wheelwright, S., Makridakis, S., & Hyndman, R. J. (1998). *Forecasting: methods and applications*. John Wiley & Sons.

Wu, L., Kong, C., Hao, X., & Chen, W. (2020). A short-term load forecasting method based on gru-cnn hybrid neural network model. *Mathematical problems in engineering*, *2020*, 1–10.

Yang, S. (2021). A novel study on deep learning framework to predict and analyze the financial time series information. *Future Generation Computer Systems*, *125*, 812–819.

Zeng, A., Chen, M., Zhang, L., & Xu, Q. (2022). Are transformers effective for time series forecasting? *arXiv preprint arXiv:2205.13504*.

Zhang, Q., Yang, L. T., Chen, Z., & Li, P. (2018). A survey on deep learning for big data. *Information Fusion*, *42*, 146–157.

Zhang, X., Wang, R., Zhang, T., Liu, Y., & Zha, Y. (2018). Short-term load forecasting using a novel deep learning framework. *Energies*, *11*(6), 1554.

# Appendices

## A. Comparison Error of Validation Real-Dataset Before Tuning

| Evaluation Error | LSTM | NBEATS | TCN | Transformer |
|---|---|---|---|---|
| RMSE | 5818.29 | 5464.36 | 5285.05 | 5530.42 |
| MAE | 2236.64 | 1266.79 | 1358.34 | 1535.72 |
| SMAPE | 1.25 | 1.3 | 1.31 | 1.31 |
| WAPE | 1.38 | 1 | 1.07 | 1.21 |

## B. Comparison Error of Validation Real-Dataset After Tuning

| Evaluation Error | LSTM | NBEATS | TCN | Transformer |
|---|---|---|---|---|
| RMSE | 5402.12 | 4728.09 | 4318.74 | 4433.02 |
| MAE | 1345.73 | 1436.25 | 1352.88 | 1396.91 |
| SMAPE | 1.24 | 1.34 | 1.2 | 1.23 |
| WAPE | 1.06 | 0.89 | 0.84 | 0.86 |

# C. Henkel Sales Data Distribution