Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Artem Moskalev

Demand forecasting for fast-moving products in grocery retail

Advanced time series regression modelling

Master's Thesis
Espoo, March 15, 2019

Supervisor: Professor Alexander Jung, Aalto University
Advisors: Tuomas Viitanen, PhD, Relex
Henri Nikula, M.Sc., Relex

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Artem Moskalev

Title:
Demand forecasting for fast-moving products in grocery retail Advanced time series regression modelling

Date: March 15, 2019 **Pages:** 129

Major: Machine Learning and Data Mining **Code:** SCI3044

Supervisor: Professor Alexander Jung

Advisors: Tuomas Viitanen, PhD
Henri Nikula, M.Sc.

Demand forecasting is a critically important task in grocery retail. Accurate forecasts allow the retail companies to reduce their product spoilage, as well as maximize their profits. Fast-moving products, or products with a lot of sales and fast turnover, are particularly important to forecast accurately due to their high sales volumes.

We investigate dynamic harmonic regression, Poisson GLM with elastic net, MLP and two-layer LSTM in fast-moving product demand forecasting against the naive seasonal forecasting baseline. We evaluate two modes of seasonality modelling in neural networks: Fourier series against seasonal decomposition. We specify the full procedure for comparing forecasting models in a collection of product-location sales time series, involving two-stage cross-validation, and careful hyperparameter selection. We use Halton sequences for neural network hyperparameter selection.

We evaluate the model results in demand forecasting using hypothesis testing, bootstrapping, and rank comparison methods. The experimental results suggest that the dynamic harmonic regression produces superior results in comparison to Poisson GLM, MLP and two-layer LSTM models for demand forecasting in fast-moving products with long sales histories. We additionally show that deseasonalization results in better forecasts in comparison to Fourier seasonality modelling in neural networks.

Keywords: Machine learning, statistics, time series, demand forecasting, dynamic harmonic regression, Poisson regression, neural networks, long short-term memory, multilayer perceptron, rolling origin cross-validation, retail

Language: English

Acknowledgements

I would like to say thanks to all the great people who helped me to complete this work: professor Alexander Jung, advisors Tuomas Viitanen and Henri Nikula, as well as my other wonderful colleagues who I discussed my thesis with, who inspired me with great ideas, and explained how with great forecasts comes great reduction in food spoilage.

I am grateful to my company Relex Solutions for providing me with this opportunity to research an interesting and important problem, and acquire valuable insights into the world of grocery retail.

The research is dedicated to my mother, Olga, who helped me through many difficult times in my life, and greatly encouraged me during my studies. Thank you, Mom.

Espoo, March 15, 2019

Artem Moskalev

Abbreviations and Acronyms

SKU	Stock keeping unit
PL	Product-location
ARIMA	Autoregressive integrated moving average
NN	Neural network
HW	Holt-Winters
SARIMA	Seasonal autoregressive integrated moving average
MLP	Multilayer perceptron
MAPE	Mean absolute percentage error
RMSE	Root mean square error
ANN	Artificial neural network
MAE	Mean absolute error
MdAPE	Median absolute percentage error
RNN	Recurrent neural network
LSTM	Long short-term memory
MSE	Mean squared error
SVM	Support vector machines
CV	Cross-validation
i.i.d.	Independent identically distributed
r.v.	Random variable
OLS	Ordinary least squares
SSE	Sum of squared errors
GLM	Generalized linear model
ARMA	Autoregressive moving average
ARIMA	Autoregressive integrated moving average
ReLU	Rectified linear unit
SGD	Stochastic gradient descent
MBGD	Mini-batch gradient descent
MLE	Maximum likelihood estimation

Contents

Chapter 1

Introduction

In the past few decades, the capabilities of computer systems have increased dramatically. The problems, where the meaningful insight has recently been impossible due to insufficient amount of data, or lack of computational resources, can now be solved with the help of the techniques from the fields of machine learning, and statistics. New algorithms [55], optimization techniques [51], and tools [1, 80] make the inference from big data possible within reasonable time.

Very often data is collected through time, and represented as a sequence of time-indexed data points. Such sequences are called time series, and frequently encountered in science and business. In the business settings, exploratory time series analysis can be used to find interesting patterns in consumer behaviour, hidden dependencies between events, or to assess how particular business decisions influenced the company growth, sales, or some other processes. Very often the time series data is also used for forecasting. Other common tasks in time series analysis include visualization, anomaly detection, classification, clustering, segmentation, and dimensionality reduction.

1.1 Retail Sales as Time Series

One interesting application of time series forecasting is the prediction of future sales in retail, or demand forecasting. The business can save money by ordering the right amount of products for each of its stores [25]. In particular, if too much of some product is ordered, and there is not enough demand at the shop, then the product stays on the shelves until the expiry date, when it has to be removed, which results in money loss. At the same time, if there is not enough of some product at some outlet, the customers cannot purchase

more of it, and the retailer loses its potential earnings. Thus, the problem of accurate demand forecasting is of primary importance to the retail business.

In retail stores, each product can be tagged with the location, where it is kept. Hence, the inventory of a retailer can be listed as a set of product-locations [82], or stock keeping units (SKU). Each product-location has an associated sales history on a weekly, daily, hourly, or some other level. However, too much granularity for the sales time series data is rarely useful, since ordering is not usually done more than once a day. Thus, the sales histories and forecasts are usually required on a daily level.

The forecasts can be produced either for each product-location separately, or for some aggregate level, such as a product group, or a group of locations. The selection of the forecast aggregate level depends on the suspected demand generation process, and the level on which the decision about the forecast usefulness is made [82]. In supply chains, SKUs are often arbitrarily classified by the amount of sales. It is easy to notice that product-locations can be either fast-moving, characterized by very high stock turnover and frequent purchases, or slow-moving with intermittent demand, where a lot of days have no sales at all. Forecasting of slow-moving products presents difficulties of its own, since it is hard to estimate parameters from a scarce sales history which is a hallmark of slow-moving product-locations.

Sales histories often have multiple seasonalities, as well as abrupt level shifts. Additionally, various calendar effects, such as the number of weekends in a month or public holidays like Christmas and Easter, can influence the demand dramatically [56]. These effects can interfere with seasonalities in a non-linear way, giving rise to unusual demand patterns. Moreover, sometimes an event like a stockout can happen. It occurs when the product is sold out at a retail shop. The stockouts have a negative impact on the forecasts of the affected product-locations, since the sales histories, that contain them, do not represent the real demand anymore [70].

The sales histories in grocery retail are often represented as time series of counts since many products are sold by piece. For the products, that are sold by weight or volume, the sales history is usually represented by fractional numbers with up to 2 decimal digits of precision. Sales time series often have external regressors which are recorded and supplied along with the sales data. These additional variables can be viewed as time series themselves, and may include information about the weather, advertising campaigns, or other events. Calendar effects, weekdays, seasonalities, and other significant implicit occurrences are often encoded as artificially engineered features.

There are multiple models and methods which are used in demand forecasting in retail. They can be broadly divided into qualitative and quantitative methods [48]. Qualitative methods are also known as judgmental

forecasting. They are most often used in situations when history sales data is not available or scarce, and the forecast has to be based on the informed opinion of an expert. On the other hand, quantitative methods are used in situations when the sales history is readily available, and include a broad range of statistical and machine learning procedures. Very often, after applying an appropriate quantitative method, forecasts are adjusted using judgment.

1.2 Contributions

It is important to determine beforehand the length of the computed forecasts. The forecast horizon, or length, depends on the practices established in a particular supply chain, and the type of a product. For fast-moving products with stable demand, precise forecasting on daily level is crucial since relevant product-locations have many sales, and the mismatch in the demand and supply results in big losses of profits. The goal of this thesis is to evaluate the performance, as well as the advantages and disadvantages of 6 quantitative methods (2 statistical and 4 machine learning models) for fast-moving product demand forecasting on a daily level. The experimental part of this paper is based on the real sales histories received from a grocery retail supermarket in Europe. In addition to the sales data, the recorded time series also have the information about the promotions and weather. We limit our attention to the models which can handle time series with regressors. The model forecasts are produced and evaluated on the product-location level. The evaluated forecast horizon is 7 days, and the forecast preparation frequency is assumed to be approximately once in a week, so that the forecasts do not overlap.

The secondary objective of this thesis is to investigate the implications of multiple seasonality, external regressors, large fluctuations in demand, as well as stockouts in sales histories. We present two approaches to multiple seasonality modelling in demand forecasting, and compare them in neural network models. We show a method to handle stockouts appropriately, so that the compared models accurately account for the out-of-stock situations in the past, and the forecasts accurately represent the future demand. In the experimental part, we additionally evaluate the effect of large fluctuations on forecast accuracy.

The thesis aims to establish and describe the methodology of forecasting model comparison. We provide a detailed description of the steps that should be taken to produce the required forecasts, how the forecasts should be evaluated with respect to a large number of product-location time series, and the details of the model comparison. This work shows a new approach to

compare multistep forecasting models based on a large number of time series with regressors, using a series of cross-validation steps, and nonparametric hypothesis testing. Our attention is focused exclusively on daily multistep forecasts of fast-moving products in grocery retail, literature on which is scarce.

1.3 Thesis Structure

The thesis chapters are briefly described as follows:

- Chapter 1 presents the specifics of time series analysis and forecasting in retail, states the main problem and goals of the thesis.
- Chapter 2 introduces the key concepts of statistics and machine learning, that are necessary to understand the further discussion.
- Chapter 3 is a broad review of related applied research in retail demand forecasting.
- Chapter 4 presents the theoretical and practical details of the methods used in the experimental part of this thesis.
- Chapter 5 summarizes the experimental part, and presents it as a sequence of steps. This chapter also briefly lists the technologies and tools used in the experiments.
- Chapter 6 states the results achieved in the experimental part in the form of tables and graphs with summaries and descriptions.
- Chapter 7 touches upon the experimental process in a critical manner, lists the advantages and disadvantages of the used methods, and gives another summary of the results.
- Chapter 8 gives the final brief statement on the results of demand forecasting model comparison for fast-moving products.

Chapter 2

Background

2.1 Machine Learning and Statistics

Machine learning is a field of study that is concerned with statistical methods and algorithms used by computer systems to build models from data in order to make decisions and solve problems without being explicitly programmed to [11, 53]. As such, machine learning is closely related to statistics. The two fields differ in their modelling and decision making philosophy, as well as practitioner communities and research focus.

Statistics places great emphasis on assumptions about the process generating the data, and then inference, hypothesis testing, and asymptotic properties are studied. Breiman [12] notes, that the biggest problem with this approach is that the nature does not follow predefined assumptions, and too much emphasis on artificial models may result in wrong conclusions. In contrast to this approach, machine learning treats the model as a black box not paying as much attention to assumptions.

Many tools used in both fields are the same. Machine learning explores the applications of linear regression models, Bayesian inference and other tools, originating in statistical literature, to computationally hard problems. At the same time, tools that were created in the machine learning community, such as regression and classification trees, support vector machines, and neural networks are researched and explained from the statistical viewpoint.

In terms of application, machine learning emphasizes prediction tasks over inference, confidence intervals or asymptotic analysis. In this work, we do not shy away from using statistical procedures that are not widely used in machine learning research, such as hypothesis testing, but the emphasis is on prediction and out-of-sample model performance comparison using as few assumptions as possible, which explains some modelling choices further.

2.2 Machine Learning Basics

Machine learning problems can be roughly presented as either supervised or unsupervised. In supervised learning problems, data is presented as a collection of records, called data points, and each data point has two parts: input (feature vector, explanatory variables, independent variables, covariates, or regressors) and output (label, response, target variable, or dependent variable). Suppose that data labels are related to their features by some unknown process $\boldsymbol{y} = f(\boldsymbol{x})$, where $\boldsymbol{y}$ is the target variable, and $\boldsymbol{x}$ is the feature vector. The purpose of supervised machine learning is to find a hypothesis function $\boldsymbol{y} = h(\boldsymbol{x}; \boldsymbol{\theta})$ from the hypothesis space $\mathcal{H}$, such that the function $h(\boldsymbol{x}; \boldsymbol{\theta}) \in \mathcal{H}$ approximates the true underlying process $f(\boldsymbol{x})$ as close as possible given some error measure, $\mathcal{L}(f(\boldsymbol{x}), h(\boldsymbol{x}; \boldsymbol{\theta}))$, called the loss function. In statistical learning theory, the optimized quantity is called the "risk", which is the expectation of the loss function over the true data-generating distribution. However, the true distribution is never known, and the quantity that is optimized in reality is the empirical risk, which is the approximation to the true data generating process based on the training set. The hypothesis space is a collection of functions that can approximate the true data generating process $f(\boldsymbol{x})$ under the initial restrictions of the used learning procedure. Supervised learning problems can be further subdivided into classification and regression problems. In classification problems, the output $\boldsymbol{y}$ of the process $\boldsymbol{y} = f(\boldsymbol{x})$ is discrete, so each feature vector $\boldsymbol{x}$ is associated with one category from a finite set. In regression problems, the output variable $\boldsymbol{y}$ is continuous, such that $\boldsymbol{y} \in \mathbb{R}$.

Unsupervised machine learning is applied to the data which has no labels. Such data often has interesting hidden patterns which can give insight into the problem at hand. Data mining is a large field related to machine learning that explores learning algorithms to discover various patterns in unlabelled datasets. Common unsupervised learning problems include clustering, density estimation, and dimensionality reduction for the purpose of visualization [11]. Other two types of machine learning are semi-supervised and reinforcement learning. Semi-supervised learning deals with the data which is only partially labelled, while reinforcement learning involves the interaction of an artificial intelligence agent with the environment in order to receive rewards for its actions and learn interactively.

In supervised machine learning, the best hypothesis has to be learned (estimated) from the training data before the predictions for the new data can be made. The hypothesis space limited to a certain set of parameters and shapes is further referred to as a model. Finding the best hypothesis is the

same as fitting or training the model. In order to find the best hypothesis, the training data is usually split into the training, validation, and test sets. The training set is used to estimate the parameters $\boldsymbol{\theta}$ of the model $h(\boldsymbol{x};\boldsymbol{\theta})$. However, some of the model parameters, called hyperparameters, have to be set manually before the training takes place. The validation set is used to find the optimal hyperparameters of the model, or to evaluate the model out-of-sample performance. The test dataset, or holdout dataset, is used specifically for reporting. No parameter learning, or model selection should be performed on the test set. The training/validation/test set division can be roughly 50%/25%/25% of the total training data available [29]. The more labelled data is available, the larger the training set can be. If the labelled dataset is very large, the division can as well be 95%/2.5%/2.5%. However, the numbers vary from application to application, depend on the problem, and on the person using the procedure.

2.2.1 Model Training

In order to find a suitable set of parameters $\boldsymbol{\theta}$ for the hypothesis $\boldsymbol{y} = h(\boldsymbol{x};\boldsymbol{\theta})$, the model has to be trained with the labelled data. As stated above, the training part of the dataset is specifically allocated for this purpose. The set of suitable parameters $\boldsymbol{\theta}$ is chosen according to some loss (cost) function $\mathcal{L}$. There is a variety of loss functions available for supervised machine learning. Regression problems often use the mean squared (quadratic) or absolute loss, while classification problems can employ mean squared, cross-entropy, or hinge loss [29]. The most desirable qualities of a loss function are continuity and differentiability. For example, the quadratic loss function is both continuous and differentiable, while the absolute loss is continuous everywhere, but not differentiable at the origin. The mean squared loss for a dataset of N points can be defined as follows

$$\mathcal{L}(f(\boldsymbol{x}), h(\boldsymbol{x};\boldsymbol{\theta})) = \frac{1}{N}\sum_{i=1}^{N}(y_i - h(\boldsymbol{x}_i;\boldsymbol{\theta}))^2 \tag{2.1}$$

After the loss function is specified, some optimization procedure has to be employed to find the set of parameters $\boldsymbol{\theta}$ that minimizes the cost function value. There is a broad range of procedures used for this task, such as the gradient descent and its variations, the family of newton methods, expectation maximization and other algorithms. In the simplest cases, a closed form solution of the loss minimization can be found, and the parameters of the model can be estimated in one step. The optimal set of parameters defines the best hypothesis function $h(\boldsymbol{x};\boldsymbol{\theta}_{\text{final}}), h \in \mathcal{H}$.

2.2.2 Model Selection

Many models have hyperparameters related to regularization, model class complexity (number of neurons, trees, polynomial degree, etc.), optimization routine (learning step, momentum, number of restarts), or feature transformations (windows sizes, differencing order in time series, etc.). In order to properly set these manually selected parameters, a validation set can be allocated. Hyperparameter selection can also be done with the help of cross-validation which is discussed further. The process of hyperparameter tuning is a part of model selection.

The procedure of hyperparameter selection involves hyperparameter vectors $\boldsymbol{\xi}_i \in \Xi$ defined for the hypothesis subspace $h(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\xi}_i) \in \mathcal{H}(\boldsymbol{\xi}_i)$. The vector $\boldsymbol{\xi}_i$ is set before the loss function is optimized on the training data to find the best parameters $\boldsymbol{\theta}_{\text{final}}$ for $h(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\xi}_i)$. Then the performance of the model $h(\boldsymbol{x}; \boldsymbol{\theta}_{\text{final}}, \boldsymbol{\xi}_i)$ is assessed on the validation set. The metric used to measure the model performance on the validation set is often different from the loss function used in the training phase. The training-validation two-step procedure is iterated by trying $\boldsymbol{\xi}_1, ..., \boldsymbol{\xi}_n \in \Xi$ one-by-one, retraining the model, and then assessing its performance on the validation set. After all possible combinations of hyperparameters in the set Ξ have been exhausted, the vector $\boldsymbol{\xi}_i$, which gives the best performance on the validation set, is selected as the best set of hyperparameters, or the best model. The set Ξ, which we further call the **hyperparameter space**, is defined by the practitioner, and is often generated using the grid search, random search, or some other sophisticated procedure for hyperparameter selection like Gaussian processes.

One common procedure for the automatic hyperparameter space generation is the grid search. It involves defining a range for each model hyperparameter, which is split into a predefined number of equidistant points. We create the hyperparameter space Ξ with the help of a Cartesian product. Suppose that each hyperparameter i from the set of hyperparameters of size N has produced a set of points S_i for its respective range. Then, we define Ξ in the grid search as

$$\Xi = S_1 \times S_2 \times ... \times S_{N-1} \times S_N = \bigtimes_{1 \leq i \leq N} S_i \tag{2.2}$$

If each hyperparameter is thought of as an axis in the N-dimensional space, then each vector $\boldsymbol{\xi}_i \in \Xi$ represents a coordinate set, which uniquely identifies one particular model. The grid search is used extensively through all areas of machine learning due to its simplicity. The grid search does not require as much intuition and judgement as the manual parameter selection, since it only needs to know the hyperparameter ranges, and how far the points on each axis are from each other (grid spacing).

Validation sets, and cross-validation in general, are a typical way to perform model selection in machine learning community. In statistical community, models are often selected based on information criteria. These quantities include Akaike information criteria (AIC), deviance information criteria (DIC), widely applicable information criteria (WAIC), as well as Bayesian information criteria [34]. These measures are different from cross-validation since they are computed on the training set after the model has been fit, and many of them cannot be used for out-of-sample performance estimation across multiple datasets and model classes [47].

Among the information criteria, AIC is commonly used in time series analysis and has the following form [34]:

$$\text{AIC} = 2k - 2\log p(y|\hat{\boldsymbol{\theta}}_{\text{mle}}) \tag{2.3}$$

where k is the number of the model parameters, and $p(y|\hat{\boldsymbol{\theta}}_{\text{mle}})$ is the estimate of the maximum likelihood of the trained model. The model is better, if its AIC is smaller. Thus, this measure penalizes models with a large number of parameters. It is also evident from the formula, that in order to assess this quantity, the model has to have some assumed probability distribution $p(.)$. However, for many models in machine learning, such as neural networks or SVMs, there is no associated probability distribution, so the computation of information criteria is not straightforward (these models have to be given a probabilistic interpretation). Moreover, the probability distribution p determines the type of error which can be computed on the test set given the selected model. For example, AIC for Gaussian maximum likelihood will return the optimal model for the mean squared error, while it might not be the optimal model for MAPE or MAE [47]. There are many other issues with the AIC and its variants that make the cross-validation in machine learning more popular than these information criteria. However, when applicable, the information criteria can usually be computed much faster than the cross-validation which often becomes the decisive factor in favour of AIC and other related methods.

2.2.3 Bias-Variance Tradeoff

An important phenomenon that often arises in model selection process is overfitting. It means that the trained model has a low error on the training set, but its performance is much worse when it tries to predict new data points. In other words, the discovered model does not generalize well. Overfitting happens as a part of the bias-variance tradeoff. Suppose that we have a particular training dataset $\boldsymbol{\mathcal{D}}$, and a hypothesis $\hat{f}(\mathbf{X}; \boldsymbol{\mathcal{D}})$ that was estimated on the dataset $\boldsymbol{\mathcal{D}}$. We can relate the labels of the data points in $\boldsymbol{\mathcal{D}}$

to the features $\mathbf{X}$ as $Y = f(\mathbf{X}) + \epsilon$, where $f(\mathbf{X})$ is the true data generating function, and ϵ is a source of noise, or error, in the dataset, such that $\mathrm{E}\big[\epsilon\big] = 0, \mathrm{Var}\big(\epsilon\big) = \sigma^2$. It turns out that the expected generalization error for some given feature vector $\boldsymbol{x}_0$ can be decomposed as follows [29]:

$$\mathrm{E}_{\{Y,\boldsymbol{\mathcal{D}}\}}\big[(Y - \hat{f}(\boldsymbol{x}_0; \boldsymbol{\mathcal{D}}))^2\big] = \sigma^2 + \mathrm{Bias}^2\big(\hat{f}(\boldsymbol{x}_0; \boldsymbol{\mathcal{D}})\big) + \mathrm{Var}\big(\hat{f}(\boldsymbol{x}_0; \boldsymbol{\mathcal{D}})\big) \quad (2.4)$$

The bias and variance of the suggested hypothesis $\hat{f}(\mathbf{X}; \boldsymbol{\mathcal{D}})$ are computed over all possible random training datasets $\boldsymbol{\mathcal{D}}$. This equation allows us to analyze the error incurred for a particular test sample $\boldsymbol{x}_0$. Since the term σ^2 is irreducible, and the expected generalization error is constant, the decrease in the bias will increase the variance, while the decrease in the variance will increase the bias. High bias in the equation means that the difference between the estimator expectation $\mathrm{E}\big[\hat{f}(\boldsymbol{x}_0; \boldsymbol{\mathcal{D}})\big]$ and the true parameter $f(\boldsymbol{x}_0)$ is large. The variance part indicates how much the estimate $\hat{f}(\boldsymbol{x}_0; \boldsymbol{\mathcal{D}})$ differs depending on the training dataset.

Overfitting occurs when the bias of the model is too low while the variance is high. It means that the model can adjust too much to insignificant fluctuations in the training set data, which leads to a poor generalization capability. In this situation, even though the expectation of the estimator is close to the true value, the scatter $\mathrm{Var}\big(\hat{f}(\boldsymbol{x}_0; \boldsymbol{\mathcal{D}})\big)$ is large, and it is probable that the given model $\hat{f}(\boldsymbol{x}_0; \boldsymbol{\mathcal{D}})$ trained on $\boldsymbol{\mathcal{D}}$ will have a significant deviation from its mean. In practice, high bias is associated with rigid, inflexible, or overly simplistic models, while high variance is on the other hand attributed to complex models with many degrees of freedom. Even though overfitting is a problem, choosing an insufficiently complex model also results in poor predictions, since the bias is high. Usually, this tradeoff can be visualized as the error achieved by the model versus the model complexity as in Figure 2.1.

It is evident from Figure 2.1 that the best model for prediction is neither too simple, nor too complex. While the model complexity can always increase, the test error starts eventually growing. Thus, it is important to measure how well the model generalizes. The standard way to do it is to use the validation dataset in conjunction with regularization. After fitting the model on the training set, its performance is measured on the validation set. If the model performance is unsatisfactory, and the model is too simplistic, a more complex alternative is tried. However, if overfitting is suspected, it can be prevented by either choosing a simpler model, or tuning the regularization hyperparameters.

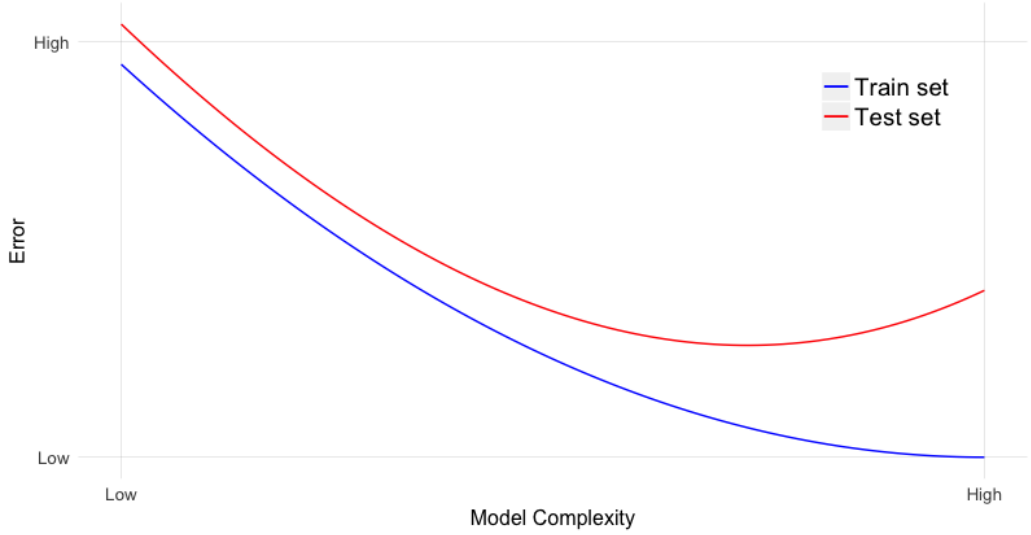

Figure 2.1: Bias-variance tradeoff

2.2.4 Model Evaluation

After the model selection has taken place, it is important to assess the performance of the selected model and compare it to other models if any. For this purpose, a test set is usually allocated. After the model selection stage, the training and validation sets are combined into one, and the final model is trained on the combined dataset with the best hyperparameters $\boldsymbol{\xi}_{\text{final}}$ to obtain $h(\boldsymbol{x}; \boldsymbol{\theta}_{\text{final}}, \boldsymbol{\xi}_{\text{final}})$. The performance of the model is often checked on the test set using the same metric that was used on the validation set during model selection. This procedure, also known as the holdout method, allows to assess the out-of-sample performance of the model.

Another way to assess the out-of-sample performance of a model involves using cross-validation (CV). The CV is a procedure that splits the dataset into multiple equally sized parts, called folds, and then selects one fold as a validation/test set, while merging all other folds into 1 training set. This procedure is repeated with each of the folds exactly once. After all folds have been used as the test set, the final accuracy measure is calculated as an average across all folds. The holdout method can be viewed as the CV with 1 fold.

There are many types of the cross-validation procedure in the literature. When the data is split into k folds, the procedure is called k-fold cross-validation. An extreme example is when k equals the number of data points in the dataset. In this case, it is called the leave-one-out cross-validation

(LOO-CV). The number of folds has to be determined manually, but the estimate can be biased if the number of folds is small [52]. A stratified k-fold cross-validation is a procedure which splits the data into k folds, and ensures that the distribution of labels in each fold is approximately the same as in the whole dataset. Stratification allows the cross-validation to have a lower bias.

2.3 Statistical Inference

Each experimental set of data is usually a subset of a much larger collection of data, called the population, that could not be obtained due to various restrictions. Thus, any experimental dataset represents only a part of all possible observations. The purpose of statistical inference is to discover the true data generating distribution or the properties of the population given a sample (set of observations) from the population. For example, in time series analysis one can analyze a sequence of observations to arrive at the parameters of the process driving it, or given the sales history, we can try to determine if the amount of sales on Wednesdays is equal to any other day of the week, or people tend to buy more products on Wednesday. The concept of statistical inference is tightly connected to the concept of a model, which represents the beliefs about the population distribution. A statistical model can be either parametric or non-parametric. Informally, parametric statistical models can be described with a finite set of parameters, while nonparametric models cannot be parameterized by a finite number of parameters [73]. Further, we discuss two common statistical inference procedures: hypothesis testing and bootstrap.

2.3.1 Hypothesis Testing

Hypothesis testing is a well-established inference method in traditional statistics. This procedure involves testing some theory, or null hypothesis H_0, against an alternative theory, H_1, where the evidence is provided by the sample data [73]. If there is enough evidence in the data, then the null hypothesis is rejected. If not, then the null hypothesis is retained.

More formally, given a set of possible parameter values Θ, we partition it into two disjoint subsets Θ_0 and Θ_1. The null and alternative hypotheses for the parameter $\theta \in \Theta$ can then be stated as follows [73]:

$$\begin{aligned} \mathrm{H}_0 : &\quad \theta \in \Theta_0 \\ \mathrm{H}_1 : &\quad \theta \in \Theta_1 \end{aligned}$$

We define an observable random variable $T(\mathbf{X})$, which is the function of a possible data sample $\mathbf{X}$, and call it a test statistic. After the sample has been collected, we can compute the statistic realization $t_{obs} = T(\mathbf{X}_{obs})$ for the observed dataset $\mathbf{X}_{obs}$. The p-value p_{obs} is the probability that the test statistic $T(\mathbf{X})$, under the assumption that the null hypothesis is true, is at least as extreme as the observed test statistic t_{obs}. We define a significance level $\alpha \in (0, 1)$ before the test, and assume that the null hypothesis is rejected if the observed p-value $p_{obs} < \alpha$. The typical significance levels are 5%, or 1%. There are 3 types of hypothesis tests: right-tailed, left-tailed, and two-tailed. These tests have the following null and alternative hypotheses:

$$\text{Two tails}: \quad \mathrm{H}_0 : \theta = \theta_0 \quad \text{vs} \quad \mathrm{H}_1 : \theta \neq \theta_0 \tag{2.5}$$

$$\text{Left tail}: \quad \mathrm{H}_0 : \theta \geq \theta_0 \quad \text{vs} \quad \mathrm{H}_1 : \theta < \theta_0 \tag{2.6}$$

$$\text{Right tail}: \quad \mathrm{H}_0 : \theta \leq \theta_0 \quad \text{vs} \quad \mathrm{H}_1 : \theta > \theta_0 \tag{2.7}$$

Given a null-hypothesis H_0, and the test statistic T, the p-values for each type of the test can be defined as

$$\text{Two tails}: \quad p_{obs} = 2 * \min(P(T \geq t_{obs}|\mathrm{H}_0), P(T \leq t_{obs}|\mathrm{H}_0)) \tag{2.8}$$

$$\text{Left tail}: \quad p_{obs} = P(T \leq t_{obs}|\mathrm{H}_0) \tag{2.9}$$

$$\text{Right tail}: \quad p_{obs} = P(T \geq t_{obs}|\mathrm{H}_0) \tag{2.10}$$

Since the test statistic T is a random variable, there is some variation associated with it, and it might be possible that even though the null hypothesis is true, the observed t_{obs} has $p_{obs} < \alpha$, and the null hypothesis is rejected. Rejecting the null-hypothesis H_0, when it is true, is an example of Type I error. On the contrary, keeping the null hypothesis H_0 when H_1 is true, is the type II error.

Before applying a given hypothesis test, a number of initial conditions must be met. These conditions, called statistical assumptions, directly influence the test statistic. The assumptions can be related to the distribution of the observations in the sample (i.i.d. observations, normal errors, etc.), or to the sampling procedure (sampling method, size of the population, size of the sample, etc.). Hypothesis tests can be either parametric or nonparametric depending on the task and the data.

2.3.2 Bootstrap

Bootstrap is a nonparametric method based on sampling with replacement which is used for computing standard errors and confidence intervals for sample estimates. The non-parametric bootstrap is a valuable tool because it can be used with minimal assumptions: it only requires that all observations in the sample should be independently and identically distributed (i.i.d.). The bootstrap is an implementation of the non-parametric maximum likelihood, and it is especially useful when no formulas for the sample estimates are available [29].

The confidence intervals for the estimate of some population parameter $\boldsymbol{\theta}$ can be obtained with the bootstrap in multiple steps. Suppose that we have a sample of N i.i.d. observations $\mathbf{X} = \{\boldsymbol{x}_1, .., \boldsymbol{x}_N\}$ from some distribution F, and some estimator $T(\mathbf{X})$. Given some number $K \in \mathbb{N}$, the bootstrap procedure is used to identify the $100(1-\alpha)\%$ confidence interval with the following steps [73]:

1. Perform N draws with replacement from the original dataset $\mathbf{X}$. Name the new sample $\mathbf{X}^*$
2. Compute a new sample estimate $\boldsymbol{\theta}_i^* = T(\mathbf{X}^*)$
3. Repeat steps 1 and 2, K times, to obtain the sample of estimates $S^* = \{\boldsymbol{\theta}_1^*, .., \boldsymbol{\theta}_K^*\}$
4. Define the sample order statistics $\boldsymbol{\theta}_{(i)}^*$ on S^*, such that $\boldsymbol{\theta}_{(1)}^* = \min(S^*)$, $\boldsymbol{\theta}_{(K)}^* = \max(S^*)$, etc.
5. Define the $100(1-\alpha)\%$ confidence interval as a range $(\boldsymbol{\theta}_{(a)}^*, \boldsymbol{\theta}_{(b)}^*)$, where $\boldsymbol{\theta}_{(a)}^*$ is the $K\alpha/2$ order statistic, and $\boldsymbol{\theta}_{(b)}^*$ is the $K(1-\alpha/2)$ order statistic on S^*

The quality of the bootstrap estimates grows with the original sample size N, and the number of bootstrap samples K [46]. The bootstrap confidence interval estimation is a widely applicable method due to its non-parametric nature, and simplicity. It is often applied to the estimation of confidence bounds and standard errors for the mean, variance, and other higher sample moments. It can also be used to estimate the confidence interval of model forecasts. There exists a parametric version of the bootstrap too [29].

2.4 Time Series Analysis

We define a stochastic process as a collection of random variables $\{Y_s\}_{s\in S}$ indexed by a set of numbers S, which represents time. Time series is a realization of this stochastic process observed at specific times $T \subseteq S$, and defined as $\{y_t\}_{t\in T}$. Time series have a distinct order which makes them different from cross-sectional data, which has no such dependence. However, time series is a realization of some collection of random variables, and, as such, they can be analyzed with general statistical tools.

Each point y_t in a time series is a realization of some distribution $p(y_t; \boldsymbol{\theta}_t)$. In usual statistical analysis with cross-sectional data, the population mean, variance, higher moments and other quantities of interest could be reliably obtained from the data with appropriate estimators and sufficient number of samples. However, each time series is a single realization of the collection of random variables $\{Y_t\}_{t\in T}$ which effectively means that it is a single sample from a multidimensional distribution of size T. Since no reliable estimate can usually be obtained from a single data point, multiple further assumptions about the properties of the time series have to be made.

Given a stochastic process $\{Y_t\}_{t=-\infty}^{+\infty}$, let us define the expectation of a random variable at time $t \in T$ as $\mathrm{E}\big[Y_t\big]$, and the covariance between two random variables at times $t, s \in T$ as $\mathrm{Cov}\big(Y_t, Y_s\big)$. The stochastic process is called weakly stationary or covariance-stationary if the expectation, variance, and covariance between the random variables do not depend on time [13]. In other words, the first and the second moments of the distribution of Y_t remain the same through time, or for all $h \in \mathbb{Z}$:

$$\begin{aligned} \mathrm{E}\big[Y_t\big] &= \mathrm{E}\big[Y_{t+h}\big] \\ \mathrm{Cov}\big(Y_t, Y_s\big) &= \mathrm{Cov}\big(Y_{t+h}, Y_{s+h}\big) \\ \mathrm{Corr}\big(Y_t, Y_s\big) &= \mathrm{Corr}\big(Y_{t+h}, Y_{s+h}\big) \end{aligned}$$

With respect to stochastic processes, the covariance and correlation between random variables are called autocovariance and autocorrelation respectively, since the random variable Y_t correlates with its own past value in the stochastic process. We define the expectation, the autocovariance and autocorrelation as follows:

$$\mathrm{E}\big[Y_t\big] = \mu, \quad \text{for all } t \tag{2.11}$$

$$\mathrm{E}\big[(Y_t - \mu)(Y_{t-j} - \mu)\big] = \gamma_j, \quad \text{for all } t \text{ and } j \tag{2.12}$$

$$\mathrm{Corr}\big(Y_t, Y_{t-j}\big) = \gamma_j/\gamma_0 = \rho_j, \quad \text{for all } t \text{ and } j \tag{2.13}$$

The quantities 2.11, 2.12 and 2.13 can be estimated from the available time series data. The estimated autocorrelations, or sample autocorrelations,

can be conveniently summarized using autocorrelation plots. The autocorrelations between variables can also be examined after the linear dependence on the variables at shorter lags has been removed. It can be achieved with partial autocorrelation plots. Suppose that the variables in stochastic process $\{Y_t\}_{t\in T}$ have the following moments: $\mathrm{E}\big[Y_t\big] = 0, \mathrm{Var}\big(Y_t\big) = 1$, and $\mathrm{E}\big[Y_t Y_s\big] = 0, t \neq s$. Then the variables Y_t represent a weakly stationary process called a white noise process, $\mathrm{WN}(0, 1)$. Figure 2.2 depicts the sample autocorrelation plot of this process.

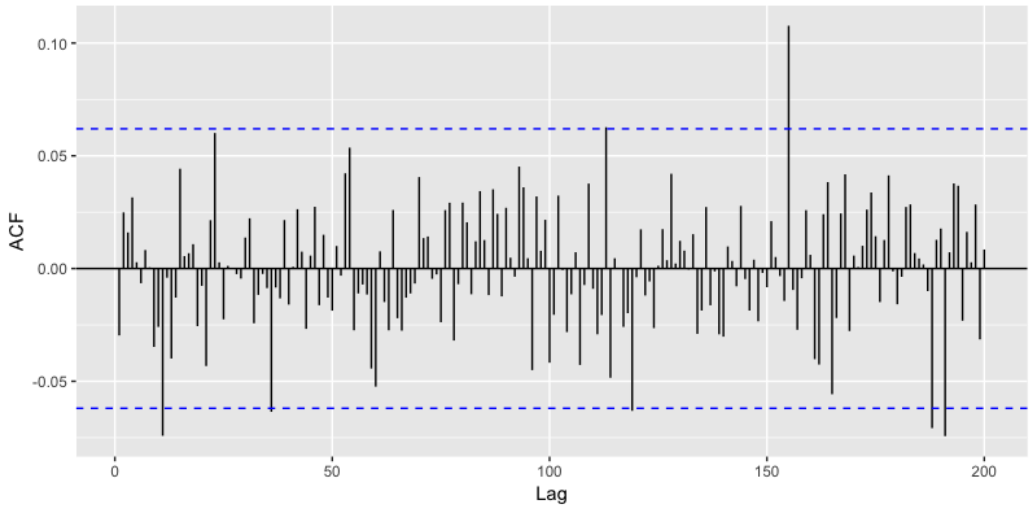

Figure 2.2: Autocorrelation plot of $\mathrm{WN}(0, 1)$

If the time series is not weakly stationary, its first or second moments depend on time, because the marginal distribution of random variables $\{Y_t\}_{t\in T}$ changes through time. Such time series often have obvious patterns, that can be exploited in data analysis. Among these patterns, the most common are trend-cycles, seasonalities, level shifts, and heteroscedasticity (change in variance). The seasonality is a systematic pattern in data which repeats itself over equal intervals. The time series data can have multiple seasonalities. The trend represents a long-term increase or decrease in time series values [48]. Very often the trend grows or decays linearly, but can also exhibit a non-linear behavior. Trends are often connected to non-systematic long-term oscillations in time series observations called cycles. The cycles differ from seasonalities because the latter always have a set frequency, while the cycles do not. Level shifts represent an abrupt change in the mean of the time series, such that $\mathrm{E}\big[Y_t\big] \neq \mathrm{E}\big[Y_s\big], t \neq s$, and $|t - s|$ is small. The

heteroscedasticity usually appears as a growing amplitude of random oscillations in time series as $t \to \infty$. One of the most common ways to overcome it is to use a data transformation such as the logarithm, or a general Box-Cox transform, on the time series.

Trend-cycles and seasonalities are systematic patterns, and can be regarded as the components of the time series. By separating these components, the following classical additive decomposition model [13, 48] can be produced:

$$\begin{aligned} Y_t &= M_t + S_t + \epsilon_t \\ \mathrm{E}\big[\epsilon_t\big] &= 0 \end{aligned} \tag{2.14}$$

where M_t is the trend value, and S_t is the seasonality addition, such that $S_t = S_{t+D}$, where D is the seasonal period, $\sum_{t=1}^{D} S_t = 0$, and ϵ_t is a remainder, or zero-mean random component, at time t. The multiplicative analog of the classical decomposition exists as well [48]:

$$\begin{aligned} Y_t &= M_t \times S_t \times \epsilon_t \\ \mathrm{E}\big[\epsilon_t\big] &= 0 \end{aligned}$$

It can be turned into the classical additive decomposition by using a log-transformation, since $\log(M_t \times S_t \times \epsilon_t) = \log M_t + \log S_t + \log \epsilon_t$. An example of an additive time series decomposition is presented in Figure 2.3.

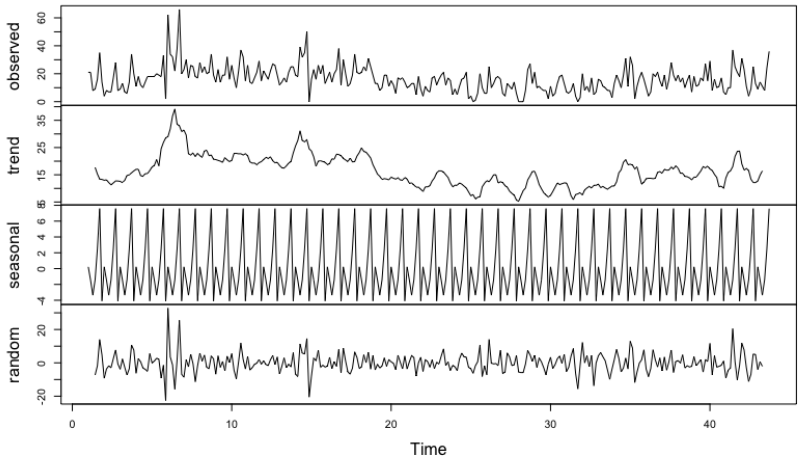

Figure 2.3: Classical additive time series decomposition.

2.5 Linear Statistical Models

In parametric statistics, strict modelling assumptions are usually made before trying to predict or describe a certain process. During the inference it is often assumed that the experimental data is generated by some unknown probability model of a certain class. A very important and widely used class of linear models has an intrinsic assumption that there is some linear relationship between population variables. The linear models are widespread in both statistics and machine learning due to their simplicity, solid theoretical background, and many theoretical guarantees which are partly due to the linearity assumption.

2.5.1 Linear Regression

Linear regression is one of the most commonly used models in statistical analysis and machine learning. It tries to model a linear relationship between the dependent random variable Y_t and a set of M explanatory variables, vector $\mathbf{X}_t^T = (\mathbf{X}_{t1}, .., \mathbf{X}_{tM})$. Each data point t in the sample is a realization of the random variable Y_t given its associated $\mathbf{X}_t$. If there is only one explanatory variable for each observation, then the regression is called a simple linear regression, while multiple linear regression is the name of the model with more than one covariate. The linear regression assumes that the random variable Y_t is related to the covariates $\mathbf{X}_t$ with the following linear relationship:

$$Y_t = \boldsymbol{\theta}^T \mathbf{X}_t + \epsilon_t \tag{2.15}$$

This model naturally has a few other assumptions. All $\mathbf{X}_t$ are considered to be non-random and the errors ϵ_t are independent from all $\mathbf{X}_t$. The regression errors ϵ_t are i.i.d. for all t and $\mathrm{E}\big[\epsilon_t\big] = 0$, $\mathrm{Var}\big(\epsilon_t\big) = \sigma^2$. The aim of the inference is to estimate the coefficient vector $\boldsymbol{\theta}$, which quantifies the linear dependence between $\mathbf{X}_t$ and Y_t.

The estimation in linear regression models can be done in many ways, but one of the reasons why the linear regression is so popular is that the most common estimation procedure called the ordinary least squares (OLS) has a closed form solution. Given the realizations y_t of Y_t and the features $\mathbf{X}_t$, the OLS minimizes the sum of squared errors (SSE):

$$\mathrm{SSE} = \sum_{t=1}^{N} (y_t - \boldsymbol{\theta}^T \mathbf{X}_t)^2$$

where the optimization is done with respect to $\boldsymbol{\theta}$. This equation can be written in the matrix form, where $\mathbf{X}$ is a matrix with the feature vectors $\mathbf{X}_t$

as its rows, and a column vector $\mathbf{Y}$, where each row is y_t:

$$\text{SSE} = (\mathbf{Y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{Y} - \mathbf{X}\boldsymbol{\theta}) \tag{2.16}$$

This function is convex with respect to $\boldsymbol{\theta}$, and thus, has one global minimum. It can be easily verified that the solution of this minimization problem can be written as:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{minimize}}\ \text{SSE} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

This estimator of the parameters $\boldsymbol{\theta}$ is unbiased which can be easily checked by denoting the column vector of random errors as $\boldsymbol{\epsilon}$, and taking the expectation of the estimator:

$$\text{E}\big[(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}\big] = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\text{E}\big[\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon}\big] = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} = \boldsymbol{\theta}$$

which proves that the OLS estimator is unbiased. A further result, called Gauss-Markov theorem, states that the OLS estimator is the best linear unbiased estimator [29]. It can also be noted, that if the errors $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$ then $Y_t \sim \mathcal{N}(\boldsymbol{\theta}^T\mathbf{X}_t, \sigma^2)$ and the OLS corresponds to the maximum likelihood estimation of the mean of Y_t.

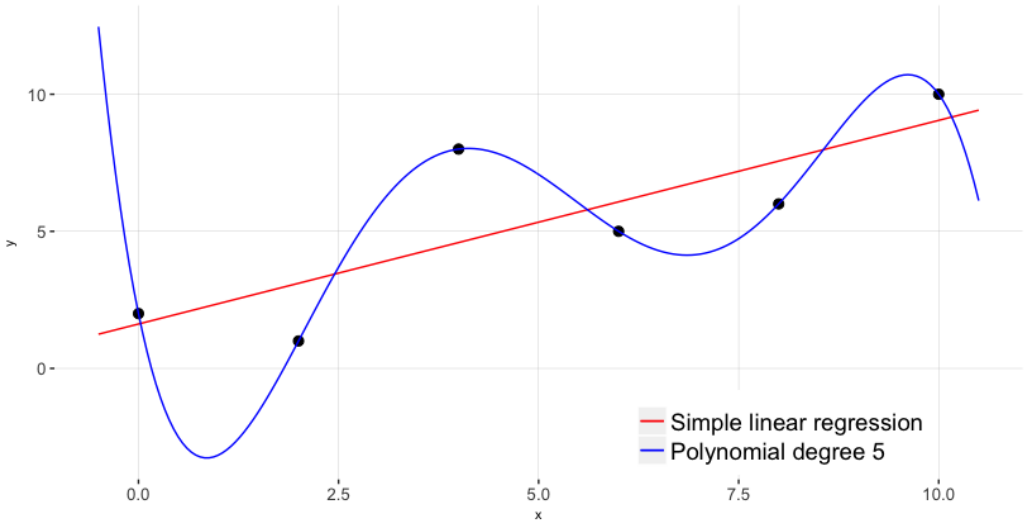

Figure 2.4: Linear regression model fit to a set of 6 data points: with and without polynomial basis functions.

As suggested by its name, the linear regression can be used to model linear relationships in data. It is often insufficient to model complex non-linear

relationships between the regressors and the response variable. Sometimes the problem of non-linearity can be ameliorated by using basis functions [11]. This solution involves mapping the features $\mathbf{X}_t$ of dimension S to some other dimension M through $f : \mathbb{R}^S \to \mathbb{R}^M$. This operation allows to extend the linear regression models to polynomials and other non-linear functions with respect to the original features $\mathbf{X}_t$. However, the regression remains to be linear in its coefficients $\boldsymbol{\theta}$, and hence the name. This method allows to decrease the high bias of simplistic linear models as demonstrated in Figure 2.4.

In Figure 2.4 the features are mapped from the original space into the space of polynomial functions. The linear regression with the polynomial features is then compared to the simple linear regression without basis functions. It is clear from the image that using a 5-degree polynomial feature transformation can achieve zero error, while simple linear regression is rigid and has a lot of bias. This result shows the importance of feature engineering.

2.5.2 Generalized Linear Models

Generalized linear models (GLM) are an extension of the simple and multiple linear regression to the cases when the response variable is assumed to follow some other distribution that normal. For example, GLMs are widely used in case the dependent variable is categorical or constrained. The linear regression with the assumption of Gaussian errors is just a specific instance of the GLM. The GLM theory provides the unified framework for the linear, Poisson and logistic regression [59]. Moreover, any distribution that belongs to the exponential family can be used in the GLM regression. The class $\mathcal{F}$ of probability distributions belongs to the exponential family if all of its members can be transformed into the following form [28]:

$$p_Y(y; \theta, \phi) = \exp\Big[\frac{y\theta - b(\theta)}{a(\phi)} + c(y, \phi)\Big] \tag{2.17}$$

where y is an observation from the distribution p_Y that belongs to the class $\mathcal{F}$. θ and ϕ are distribution parameters. ϕ is a dispersion parameter which is usually known and related to the variance of the distribution p_Y. The functions $a(.), b(.),$ and $c(.)$ are known and depend on the class $\mathcal{F}$ of the distribution. Parameter θ is a canonical parameter for the exponential family $\mathcal{F}$, if it is a function of the expectation, $\theta = g_c(\mathrm{E}\big[Y\big])$. The function $g_c(.)$ is called the canonical link function if it does not depend on the dispersion parameter ϕ.

Given the response variable Y_t and a set of regressors $\mathbf{X}_t$, the GLM can be introduced in terms of 3 components as follows [28]:

1. Random component

   It is the conditional distribution $f(Y_t|\mathbf{X}_t)$ of the random variable Y_t given a vector of constant regressors $\mathbf{X}_t$. This distribution has to belong to the overdispersed exponential family [59]. By extension, all distributions in the exponential family (Normal, Poisson, Binomial, Gamma) can also represent the random component of a GLM.

2. Linear predictor

   It is a linear function of the regressor vector $\mathbf{X}_t$ that can be denoted as $\eta_t = \boldsymbol{\beta}^T\mathbf{X}_t$, where $\boldsymbol{\beta}_0$ is the intercept, and $\mathbf{X}_{t0} = 1$. The features $\mathbf{X}_t$ can include basis functions and binary variables similar to the multiple linear regression.

3. Link function

   It is a function $g(.)$ that relates the mean μ_t of the random component $f(Y_t|\mathbf{X}_t)$ to the output η_t of the linear predictor $\boldsymbol{\beta}^T\mathbf{X}_t$, such that $g(\mu_t) = \eta_t$. The link function has to be invertible.

Thus, the GLM is a probabilistic model where the distribution of the response variable depends on the linear combination of the input variables as its mean. The aim of the GLM regression is to maximize the log-likelihood function represented by the logarithm of the random component with respect to the vector of parameters $\boldsymbol{\beta}$. This log-likelihood function is not concave in general, and does not always have a simple closed-form solution like the multiple linear regression, so it has to be optimized with numerical methods. The traditional approach to optimizing common GLM log-likelihood functions is to use the iteratively reweighted least squares (IRLS) algorithm [59]. However, if the canonical link function is used, IRLS becomes similar to the Newton-Raphson (Newton's) optimization procedure [28]. The Newton's method can have problems with local maxima, but if the link function is canonical, the optimized GLM log-likelihood is always concave, so the algorithm can find the global log-likelihood optimum. The Newton-Raphson method is preferable to gradient ascent methods in applicable situations, since it is much faster [41].

The Newton-Raphson optimization method is based on the Newton's method to find the roots of a function and on the fact that the maxima and minima of a function have their derivatives equal to zero. In order to find the maximum of the log-likelihood function $\mathcal{L}(\boldsymbol{\beta})$, where $\boldsymbol{\beta}$ is a vector

of parameters, it is necessary to find the points where its gradient is zero, or $\nabla_{\boldsymbol{\beta}}\mathcal{L} = 0$. The Newton's method can help to find the roots of the gradient function. Since the gradient is a vector valued function, its first-order partial derivative is $\mathbf{H}_{\boldsymbol{\beta}} = \frac{\partial \nabla_{\boldsymbol{\beta}}\mathcal{L}}{\partial \boldsymbol{\beta}}$, where $\mathbf{H}_{\boldsymbol{\beta}}$ is the matrix of the second-order derivatives of $\mathcal{L}(\boldsymbol{\beta})$, or Hessian computed at $\boldsymbol{\beta}$. We can form a second-order Taylor expansion of the log-likelihood function around the current best parameter guess $\hat{\boldsymbol{\beta}}$ as follows:

$$\mathcal{L}(\hat{\boldsymbol{\beta}} + \Delta\boldsymbol{\beta}) = \mathcal{L}(\hat{\boldsymbol{\beta}}) + \nabla_{\hat{\boldsymbol{\beta}}}\mathcal{L}\Delta\boldsymbol{\beta} + \frac{1}{2}\Delta\boldsymbol{\beta}^T\mathbf{H}_{\hat{\boldsymbol{\beta}}}\Delta\boldsymbol{\beta} \tag{2.18}$$

This equation approximates the log-likelihood function, and it should conform to the condition of zero derivative with respect to $\Delta\boldsymbol{\beta}$ as a parameter at the maxima of the function. Setting the derivative of the Taylor expansion $\mathcal{L}(\hat{\boldsymbol{\beta}} + \Delta\boldsymbol{\beta})$ with respect to the new parameterization $\Delta\boldsymbol{\beta}$ to 0, the following result can be shown:

$$0 = \nabla_{\hat{\boldsymbol{\beta}}}\mathcal{L} + \mathbf{H}_{\hat{\boldsymbol{\beta}}}\Delta\boldsymbol{\beta}$$

which naturally leads to the definition of the Newton's method iteration in multiple dimensions:

$$\hat{\boldsymbol{\beta}}_{t+1} = \hat{\boldsymbol{\beta}}_t - s\mathbf{H}^{-1}_{\hat{\boldsymbol{\beta}}_t}\nabla_{\hat{\boldsymbol{\beta}}_t}\mathcal{L} \tag{2.19}$$

where s is a step size parameter, and t specifies the iteration number. Since the Newton's optimization routine is derived from the Taylor series expansion, it has some inherent error associated with it. Per each iteration the Newton's method gives an estimate close to the extrema of the log-likelihood function, but due to the approximation error the next suggested $\hat{\boldsymbol{\beta}}$ is not the exact optimum. Thus, Equation 2.19 has to be iteratively invoked until a sufficiently good approximation to the maximum of the GLM log-likelihood function is achieved.

As mentioned before, the Newton-Raphson method has some serious drawbacks. In general, saddle points of functions also have zero gradients. It means that if the log-likelihood function is not concave, then the algorithm can get stuck not only in local optima, but also in saddle points. The Newton-Raphson method can also have problems with slow convergence and overshooting. However, these problems can be avoided by applying the algorithm only to the GLMs with the canonical link functions. Most commonly used GLMs, such as the logistic or Poisson regression, have well-behaving log-likelihoods with a clear maximum, and the Newton-Raphson optimization method is an optimal choice.

2.5.3 Regularization in Linear Models

Both linear regression and GLM can have problems related to overfitting. An excessively flexible linear model trained on a dataset with a large number of features or with many outliers, can turn out to be too complex, and result in overfitting and bad generalization capability. For example, if polynomial basis functions are used, the regression model can fit the sample data near-perfectly. However, it is often not the desired result, since the model adjusts to the noise too much. The overfitting for the linear regression model is demonstrated in Figure 2.5, where the red line which stands for the flexible model visits all the points, but the relationship between the input and output is obviously much simpler and closer to a straight line.

The problem of overfitting is usually solved with regularization. The procedure penalizes the linear model coefficients, and results in more conservative parameter estimates. One common regularization procedure is called the ridge regression. It penalizes the linear regression model with the L_2-norm, or sum of squared coefficients. Given that the matrix $\mathbf{X}$ is a feature matrix, where each row contains the features of one data point from the training set, and $\mathbf{Y}$ is a vector of labels, the ridge regression formulation can be written as follows:

$$\mathrm{SSE}_{\mathrm{Ridge}} = (\mathbf{Y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{Y} - \mathbf{X}\boldsymbol{\theta}) + \lambda\|\boldsymbol{\theta}\|^2 \tag{2.20}$$

where $\boldsymbol{\theta}$ is the column vector of the regression coefficients. The parameter λ has to be tuned manually using the validation dataset. This equation has a closed-form solution which is straightforward to compute. The effect of the L_2-regularizer is demonstrated in Figure 2.5, where two different values of λ produce obviously different results when applied to the 5-degree polynomial linear regression. Given a large value of λ, the regularizer restricts the 5-degree polynomial to the form of an almost straight line, which is supposedly a better fit for this data.

Another common type of regularizer is the least absolute shrinkage and selection operator (LASSO). It penalizes the parameters by absolute value, or L_1-norm, and its SSE for the linear regression can be written as follows:

$$\mathrm{SSE}_{\mathrm{LASSO}} = (\mathbf{Y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{Y} - \mathbf{X}\boldsymbol{\theta}) + \lambda\sum_{i=1}^{S}|\boldsymbol{\theta}_i| \tag{2.21}$$

where $\boldsymbol{\theta}_i$ is the coefficient associated with the feature $\mathbf{X}_{ti}$, and S is the dimension of the feature vector. As with the ridge regression, λ controls the strength of the regularization and is usually chosen manually. Higher values imply stronger regularization. While the ridge regression penalizes

all the parameters equally by making them smaller, the LASSO regression tries to set some of them to zero, providing the sparse representation of the coefficient vector $\boldsymbol{\theta}$ [38].

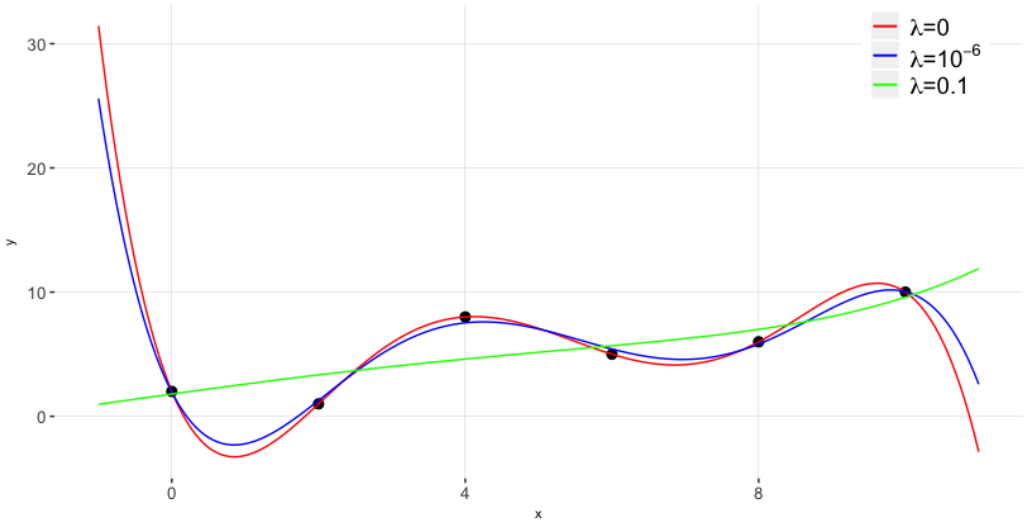

Figure 2.5: Three 5-degree polynomial linear regression models fit to a set of 6 data points with a different strength of ridge regularizer.

Generalized linear models can also be regularized like the linear regression using the L_2 and L_1 penalties. Given a vector of optimized parameters $\boldsymbol{\beta}$, the L_2 and L_1 terms are appended directly to the log-likelihood function $\mathcal{L}(\boldsymbol{\beta})$ as follows:

$$\hat{\mathcal{L}}_{\text{ridge}}(\boldsymbol{\beta}) = \mathcal{L}(\boldsymbol{\beta}) - \lambda \|\boldsymbol{\beta}\|^2 \tag{2.22}$$

$$\hat{\mathcal{L}}_{\text{LASSO}}(\boldsymbol{\beta}) = \mathcal{L}(\boldsymbol{\beta}) - \lambda \sum_{i=1}^{S} |\boldsymbol{\beta}_i| \tag{2.23}$$

Thus, the equations for the ridge GLM regression (2.22) and LASSO GLM regression (2.23) are conceptually analogous to those of the multiple regression. The term λ defines how much regularization is applied to the coefficients of the GLM model. The regularization in the GLM can be viewed as setting specific priors in Bayesian inference. The ridge regression corresponds to the multivariate normal prior, while LASSO corresponds to the multivariate Laplace prior. Like in the non-regularized GLM formulation, there are no closed form equations to estimate the coefficients $\boldsymbol{\beta}$ in the presence of the penalty term, and some numerical optimization method like Newton-Raphson or gradient ascent has to be applied.

2.5.4 Autoregressive Moving Average

Autoregressive moving average (ARMA) models are a class of linear models that can describe covariance-stationary processes. The ARMA model can be also viewed as a process with the autoregressive (AR) and the moving average (MA) parts. Both AR and MA can be separate models representing their own stochastic processes. Moving average processes are always stationary unless they are of an infinite order, while autoregressive processes can be non-stationary depending on their parameters.

In order to explain the nature of the ARMA models better, it is easier to use time series operators. The time series operator is a mapping that transforms one time series into another time series by acting on each of the time series elements [41]. Time series operators can also be applied to stochastic processes. One of such operators is the lag operator L. When applied to a time series element at t, it returns the preceding element $t-1$. Suppose that there exists a time series $\{y_t\}_{t=-\infty}^{\infty}$. If the lag operator is applied to it, then for each element t, $Ly_t = y_{t-1}$. The lag operator is commutative with multiplication, which means $L\theta y_t = \theta L y_t$, and has the distributive property, $L(y_t + y_{t-1}) = Ly_t + Ly_{t-1}$. The lag operator can also be applied iteratively, and rewritten in the exponential notation $LLy_t = y_{t-2} = L^2 y_t$.

Given a moving average stochastic process MA(q) with observable random variables $\{Y_t\}_{t\in T}$, the model can be presented in the form of a difference equation as follows:

$$Y_t = \mu + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{2.24}$$

where μ is chosen as the mean of the stationary MA(q) process, and $\epsilon_t, \forall t \in T$ follows the white noise process as per the definition in Section 2.4. The mean μ and the set of coefficients $\boldsymbol{\theta}$ represent the parameters of the moving average model, and are the target of the statistical inference. The MA(q) process can be also presented in the lag operator notation:

$$Y_t - \mu = \Big(1 + \sum_{i=1}^{q} \theta_i L^i\Big)\epsilon_t \tag{2.25}$$

The autoregressive stochastic process AR(p) involves the dependence of each time series observation on p past observations preceding it. Given the AR(p) stochastic process $\{Y_t\}_{t\in T}$, the model can be presented in the form of a difference equation as follows:

$$Y_t = c + \sum_{j=1}^{p} \phi_j Y_{t-j} + \epsilon_t \tag{2.26}$$

where c is some constant, and ϵ_t is generated by a white noise process as per Section 2.4. The variable c and a set of coefficients $\boldsymbol{\phi}$ represent the model parameters. Using the lag operator notation, we can restate the AR(p) model as follows:

$$\epsilon_t = \Big(1 - \sum_{j=1}^{p} \phi_j L^j\Big)\Big(Y_t - \mu\Big) \tag{2.27}$$
$$\mu = \frac{1}{1 - \sum_{j=1}^{p} \phi_j}$$

where μ is the expectation of the stochastic process AR(p) and can be easily found by taking the expectations on the both sides of (2.26) under the assumption that the process is weakly stationary.

Given the definitions of the MA(q) and AR(p) stochastic processes, we can define the ARMA(p, q) process as follows:

$$Y_t = c + \sum_{j=1}^{p} \phi_j Y_{t-j} + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{2.28}$$

where ϵ_t is a white noise term with variance σ^2, also called an innovation, and c is a variable related to the mean of the ARMA process as in the AR(p) process. From this presentation, it is clear that ARMA is a linear process in its coefficients $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$. The target of statistical inference is to find the population parameters $\boldsymbol{\theta}, \boldsymbol{\phi}$, c, and σ^2. We can state the ARMA(p, q) model in the lag operator notation as follows:

$$\big(1 - \sum_{j=1}^{p} \phi_j L^j\big)\big(Y_t - \mu\big) = \big(1 + \sum_{i=1}^{q} \theta_i L^i\big)\epsilon_t \tag{2.29}$$

The ARMA(p, q) process is stationary only when its AR(p) part is stationary, or in other words the polynomial $(1 - \sum_{j=1}^{p} \phi_j z^j) = 0$ has its roots outside of the unit circle. Given that the autoregressive part of Equation 2.29 is stationary, the ARMA(p, q) can be rewritten as:

$$(Y_t - \mu) = \frac{(1 + \sum_{i=1}^{q} \theta_i L^i)}{(1 - \sum_{j=1}^{p} \phi_j L^j)} \epsilon_t = \sum_{k=0}^{\infty} \psi_k L^k \epsilon_t \tag{2.30}$$

where $\psi_0 = 1$, and $\sum_{k=0}^{\infty} |\psi_k| < \infty$. It means that the stationary ARMA(p, q) process can be represented as the MA(∞) process. The similar conversion is also possible from ARMA to AR(∞) when the MA part of the ARMA process is invertible.

2.5.4.1 Forecasting

Time series forecasting amounts to predicting the value of a random variable Y_t given the history of prior observations of random variables $\{Y_t\}_{-\infty}^{t-1}$ in the stochastic process which drives the given time series. It can be shown that the optimal forecast $\hat{Y}_t$ with respect to the mean squared error $\mathrm{E}\big[(Y_t - \hat{Y}_t)^2\big]$ is the conditional expectation:

$$\hat{Y}_t = \mathrm{E}\big[\, Y_t \,|\, Y_{t-1}, Y_{t-2}, .., Y_{-\infty} \,\big]$$

However, this type of forecast is usually impossible to make since the full conditional distribution cannot be computed. Instead, the forecasts are limited to a specific class of linear functions in the form of $g(\mathbf{X}) = \boldsymbol{a}^T\mathbf{X}$, where $\mathbf{X}$ is a list of regressors, such as the past observations of the stochastic process, which includes the constant term, and $\boldsymbol{a}$ is a vector of coefficients. The linear forecast can be presented as follows:

$$\hat{Y}_t = \hat{\mathrm{E}}\big[\, Y_t \,|\, \mathbf{X} \,\big] = \boldsymbol{a}^T\mathbf{X}, \tag{2.31}$$

$$\text{s.t. } \mathrm{E}\big[(Y_t - \boldsymbol{a}^T\mathbf{X})\mathbf{X}^T\big] = \mathbf{0} \tag{2.32}$$

where the condition (2.32) set on $\boldsymbol{a}$ means that the residual between the random variable Y_t and its forecast $\boldsymbol{a}^T\mathbf{X}$ is uncorrelated with the regressor random variables $\mathbf{X}$. This condition ensures that the linear combination $\boldsymbol{a}^T\mathbf{X}$ represents the linear projection of r.v. Y_t on $\mathbf{X}$. It can be easily shown [41] that such a forecast will be the optimal linear forecast with respect to MSE $\mathrm{E}\big[(Y_t - \boldsymbol{a}^T\mathbf{X})^2\big]$. Nevertheless, the best linear forecast is at best equal to the conditional expectation:

$$\mathrm{E}\big[(Y_t - \boldsymbol{a}^T\mathbf{X})^2\big] \geq \mathrm{E}\big[(Y_t - \mathrm{E}\big[\, Y_t \,|\, \mathbf{X} \,\big])^2\big]$$

The linear forecast is much easier to compute than conditional expectation due to the property (2.32):

$$\mathrm{E}\big[Y_t\mathbf{X}^T\big] - \boldsymbol{a}^T\mathrm{E}\big[\mathbf{X}\mathbf{X}^T\big] = \mathbf{0}$$

$$\boldsymbol{a} = \mathrm{E}\big[\mathbf{X}\mathbf{X}^T\big]^{-1}\mathrm{E}\big[\mathbf{X}Y_t^T\big] \tag{2.33}$$

which shows the closed form expression to find the coefficient vector $\boldsymbol{a}$. Therefore, in order to make an optimal linear forecast, it is only necessary to find the vector of parameters $\boldsymbol{a}$, which requires the covariance matrix $\mathrm{E}\big[\mathbf{X}Y_t^T\big]$ between the forecast variable Y_t and its regressors $\mathbf{X}$, as well as the inverse of the covariance matrix $\mathrm{E}\big[\mathbf{X}\mathbf{X}^T\big]$. In order to compute the covariance matrices, the population parameters of the joint distribution of $(Y_t, \mathbf{X})$ have

to be known beforehand. If the variable Y_t is predicted using n past values of the stochastic process $\{Y_t\}_{-\infty}^{t-1}$, then $\mathbf{X} = (Y_{t-1}, .., Y_{t-n})$. In this case, an assumption is usually made about some linear model, such as ARMA, that drives the stochastic process, while the covariance matrices and the coefficient vector $\boldsymbol{a}$ are estimated based on this assumption.

The exact forecast for the ARMA process can be obtained from (2.33) by finding the required second moments. In order to achieve this, we need the ARMA population parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. Given a vector of s random variables $Y_{(t+1):(t+s)}$ to forecast, and a vector of past t values $Y_{1:t}$ of the ARMA stochastic process, we specify the column vector $\boldsymbol{v} = (\mathbf{Y}_{(t+1):(t+s)}, \mathbf{Y}_{1:t})$. Then we can compute the second-moment matrix $\Omega = \mathrm{E}\left[\boldsymbol{v}\boldsymbol{v}^T\right]$ from the population parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. Using the LDU decomposition, the closed form expressions for the coefficients $\boldsymbol{a}$ can be found [41]. Given the vector of coefficients $\boldsymbol{a}$, the forecasts can be easily produced from Equation 2.31. An effective alternative to the use of LDU factorization for exact forecasting is the Kalman filter.

2.5.4.2 Parameter Estimation

The inference in ARMA models is often done using maximum likelihood. For this procedure, some convenient error distribution has to be assumed. The random part of the ARMA(p, q) process is the white noise innovations ϵ_t, and it is often assumed that this noise is Gaussian, $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. Let us assume that for an ARMA(p, q) process, the past observations $\boldsymbol{y}_t^{past} = (y_{t-1}, .., y_{t-p})$ are known at $t = 0$, and past innovations $\boldsymbol{\epsilon}_t^{past} = (\epsilon_{t-1}, .., \epsilon_{t-q})$ are set to zeros in $\boldsymbol{\epsilon}_0^{past}$. According to the ARMA definition (2.28), the error $\epsilon_t | \boldsymbol{y}_t^{past}, \boldsymbol{\epsilon}_t^{past}$ can be expressed as

$$\epsilon_t = Y_t - c - \sum_{j=1}^{p} \phi_j y_{t-j} - \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{2.34}$$

Since the innovation ϵ_t is assumed to be Gaussian, Y_t follows the conditional normal distribution:

$$Y_t \,|\, \boldsymbol{y}_t^{past}, \boldsymbol{\epsilon}_t^{past} \sim \mathcal{N}(c + \sum_{j=1}^{p} \phi_j Y_{t-j} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}, \sigma^2)$$

Thus, using the whole history of time series observations, and the initial conditions $\boldsymbol{y}_0^{past}, \boldsymbol{\epsilon}_0^{past}$, we can rewrite the distribution of each r.v. $Y_0, .., Y_N$

as:

$$p_{Y_t}(y_t \mid \boldsymbol{y}_t^{past}, \boldsymbol{\epsilon}_t^{past}) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\big[-\frac{(y_t - c - \boldsymbol{\phi}^T\boldsymbol{y}_t^{past} - \boldsymbol{\theta}^T\boldsymbol{\epsilon}_t^{past})^2}{2\sigma^2}\big]$$
$$= \frac{1}{\sqrt{2\pi\sigma^2}}\exp\big[-\frac{\hat{\epsilon}_t^2}{2\sigma^2}\big] \tag{2.35}$$

where $\hat{\epsilon}_t$ is the realization of the innovation random variable ϵ_t, $\boldsymbol{\phi}$ represents the vector of autoregressive coefficients, and the vector $\boldsymbol{\theta}$ represents the moving average coefficients. From Equation 2.34, it is clear that each ϵ_t depends on the previous q values of the innovation process. This results in complex equations in the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ for each conditional likelihood $p_{Y_t}(.)$. The maximum likelihood estimator based on the conditional distribution $p_{Y_t}(.)$ is called the **conditional maximum likelihood function**.

Another way to estimate the parameters $\boldsymbol{\phi}, \boldsymbol{\theta}, \sigma^2$, and μ of the ARMA(p, q) model is to use the **exact log-likelihood function**, which is multivariate normal since all the innovations ϵ_t are correlated Gaussians:

$$\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu, \sigma^2) = \log p_{Y_N,..,Y_0}(y_N, .., y_0;\, \boldsymbol{\phi}, \boldsymbol{\theta}, \mu, \sigma^2)$$
$$= -\frac{N+1}{2}\log(2\pi) - \frac{1}{2}|\Omega| - \frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu})^T\Omega^{-1}(\boldsymbol{y} - \boldsymbol{\mu}) \tag{2.36}$$

where $\boldsymbol{y}$ is the vector of the observations $(Y_N, .., Y_0)$, μ is the mean of Y_t, σ^2 is the innovation variance, $\boldsymbol{\mu} = (\mu, .., \mu)$ due to the assumed stationarity of the ARMA process, and $|\Omega|$ is the determinant of the covariance matrix of the r.v. sequence $Y_N, .., Y_0$. The covariance matrix Ω depends on $\boldsymbol{\phi}$, $\boldsymbol{\theta}$ and σ^2, and in general is non-trivial to estimate. The easiest way to infer the exact maximum likelihood estimates of the ARMA(p, q) population parameters is to use the Kalman filter [41] as described further.

2.5.4.3 Autoregressive Integrated Moving Average

Autoregressive integrated moving average (ARIMA) models represent an extension of the ARMA model class. The ARIMA class of models allows to make the time series stationary with the help of differencing, and then apply an ARMA model to the transformed time series values. The ARIMA can be specified with 3 parameters as ARIMA(p, d, q), where d is the order of differencing. The parameters p and q denote the ARMA(p, q) model which is applied to the differenced time series. Given a stochastic process $\{Y_t\}_{t\in T}$, the ARIMA(p, d, q) can be written in the lag operator notation as follows:

$$\Big(1 - \sum_{j=1}^{p}\phi_j L^j\Big)\big(1 - L\big)^d\big(Y_t - \mu\big) = \Big(1 + \sum_{i=1}^{q}\theta_i L^i\Big)\epsilon_t \tag{2.37}$$

The autoregressive part of Equation 2.37 is a product of the polynomials $(1 - \sum_{j=1}^{p} \phi_j L^j)(1 - L)^d$, which can be rewritten in the form of a single lag polynomial with d unit-roots. In other words, it shows that ARIMA(p, d, q) is an instance of the ARMA$(p + d, q)$ model with d unit-roots. Therefore, each difference operation $(1-L)$ on time series $\{y_t\}_{t \in T}$ removes one suspected unit-root from the stochastic process driving the observations.

Unit-roots have to be removed because they imply non-stationarity. The unit-root in the autoregressive process indicates that the changes in the difference equation, that represents it, do not die out with time, which leads to the unstable process (which can grow or decrease for long periods of time resulting in cycles, that do not have a deterministic pattern). Many of the useful time series results do not hold with non-stationary processes. Differencing helps to transform a non-stationary autoregressive stochastic process to the stationary one by removing the unit-roots. The number of differences should equal the number of unit-roots.

If the time series can be made stationary after removing unit-roots, it is considered difference-stationary. Another type of non-stationarity is trend-stationarity. Trend-stationary processes have a pattern that depends on time (trend), but no unit-roots. The main difference is that as the time goes, the trend-stationary process converges to the trend. In practice, trend-stationary time series can be made stationary by estimating the trend and subtracting it from the time series. However, the method of trend estimation and removal does not work for difference-stationary time series, while polynomial trends of all degrees can still be removed with a sufficient number of differences [13].

2.6 Neural Networks

Artificial neural networks (ANN) are a large class of models that show state-of-the-art results in many machine learning problems, such as speech recognition, video processing, and object detection [55]. ANNs originally appeared as an attempt to model the brain activity of animals [64]. A canonical neural network can be regarded as a set of neurons connected by synapses in the brain. Thus, the most basic neural network represents a single neuron and an associated learning rule. An example of such a simplistic model is the perceptron, which is a supervised classification algorithm. The decision boundary of the perceptron is linear, so if the training data points are not linearly separable, then they cannot be classified perfectly by the perceptron.

An artificial neuron is the basic unit from which larger neural networks are constructed. It is conceptually similar to the perceptron in its functional form, but it does not represent a standalone learning algorithm. Given an

input $\boldsymbol{x}$, a vector of weights $\boldsymbol{w}$, a bias parameter b, and some activation function $\phi(.)$, the artificial neuron can be represented as a function $f(\boldsymbol{x})$ in the following way:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \phi(\boldsymbol{w}^T \boldsymbol{x} + b) \tag{2.38}$$

Artificial neurons can have a wide range of activation functions. Given sufficiently many neurons with some non-linear activation function, the ANN can have a non-linear decision boundary of any complexity. The activation function is often selected as a hyperparameter, and depends on the application. For example, the perceptron has a threshold activation function, while if a neural network has only linear activation functions, it is equivalent to the linear regression. Some popular activation functions include:

$$\text{Hyperbolic tangent: } \phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.39}$$

$$\text{Logistic: } \phi(x) = \frac{1}{1 + e^{-x}} \tag{2.40}$$

$$\text{Identity: } \phi(x) = x \tag{2.41}$$

$$\text{Rectifier: } \phi(x) = \max(0, x) \tag{2.42}$$

$$\text{Leaky rectifier: } \phi(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & \text{otherwise}, \quad \alpha > 0 \end{cases} \tag{2.43}$$

The structure of an artificial neuron is presented in Figure 2.6.

In the diagram, the light green boxes specify the components of the input feature vector, where the feature **1** corresponds to the bias parameter b. The features are multiplied by the weights to produce the linear response $a = \boldsymbol{w}^T \boldsymbol{x} + b$, and the blue box $\phi(a)$ represents some activation function applied to the linear response a of the neuron. $\hat{y}$ is the output of the activation function.

Even though a structure that consists of a single artificial neuron supplied with an optimization algorithm, like perceptron, can be trained and capable of prediction tasks, it is usually not capable of learning complex representations possible in data. In order to overcome this issue, artificial neurons are combined into larger structures, called artificial neural networks. ANNs often have a layered structure, where each layer is represented by a number of parallel artificial neurons. If the layers are connected to each other and these connections do not form cycles, the ANN is called a feedforward neural network since inputs are transformed exactly once by each neuron as the input travels throughout the network. In contrast to that, recurrent neural networks (RNN) use the outputs of some layers as inputs for the previous

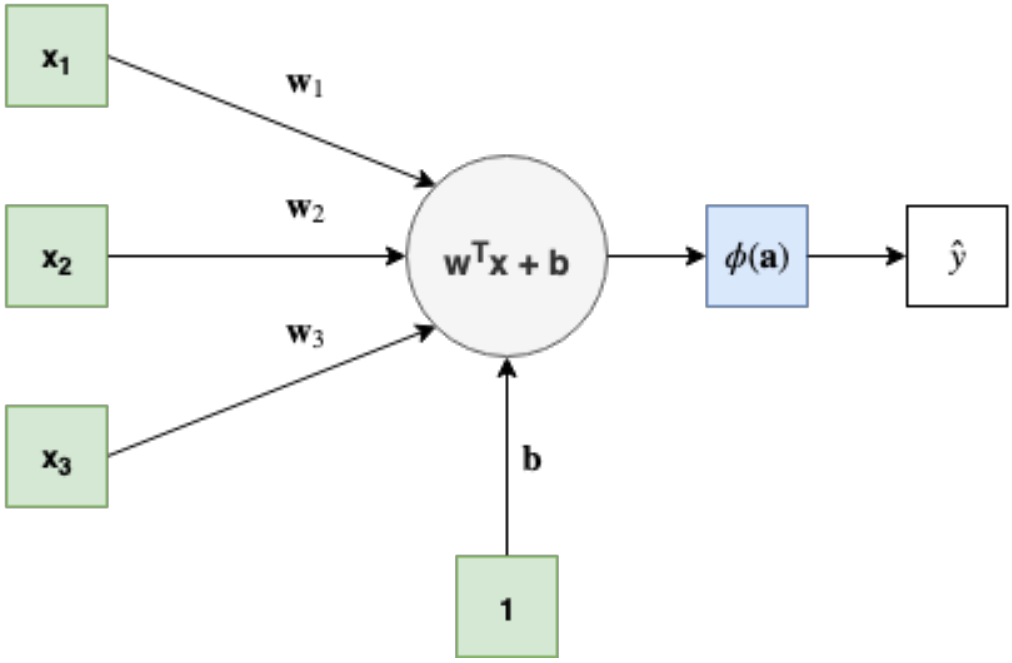

Figure 2.6: Artificial neuron with a bias term b and the input $\boldsymbol{x}$ of size 3.

layers. This creates cycles in the ANN structure. If all the neurons (nodes) in one layer are connected to each of the nodes in the next layer, then such a neural network is considered to be fully-connected.

2.6.1 Multilayer Perceptron

The most basic and "vanilla" [29] neural network is the multilayer perceptron (MLP). MLP is a feedforward fully-connected neural network which consists of at least 3 layers: input, output, and a non-zero number of hidden layers between the input and the output layers. The input layer is not a fully functional layer, since it only represents the features of the input vector $\boldsymbol{x}$, where each neuron is one variable of the feature vector $\boldsymbol{x}$. All neurons in the input layer have the bias of 0, and their activation functions are identity. After the input layer, a number of hidden layers follow, and each hidden layer i outputs the sequentially transformed vector $\boldsymbol{x}$, let us call it $\boldsymbol{y}_i$. The vector $\boldsymbol{y}_i$ is then fed as input to each neuron of the next layer $i + 1$, and so on. The output layer produces the final prediction $\hat{\boldsymbol{y}}$, that can be either a single number or a vector.

An MLP with two hidden layers is presented in Figure 2.7. In the figure, the input of the MLP is a vector $\boldsymbol{x}$, which also represents the input layer.

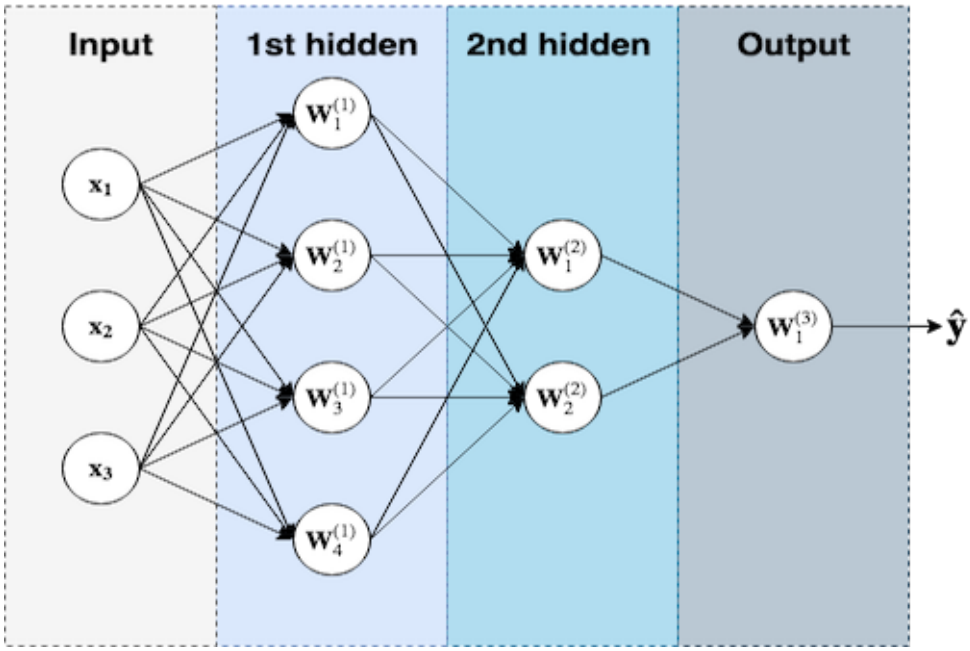

Figure 2.7: Four-layer MLP acting on a 3-dimensional feature vector $\boldsymbol{x}$.

It is fed into the first hidden layer. The weights of each neuron i in the first hidden layer are denoted as $\mathbf{W}_i^{(1)}$. As it is evident from the figure, each neuron of the first hidden layer receives the full copy of the input vector $\boldsymbol{x}$. Next, the output of the first hidden layer $\boldsymbol{y}_1 \in \mathbb{R}^4$ is fed to each node of the second hidden layer, where each neuron j has its set of weights $\mathbf{W}_j^{(2)}$. The second hidden layer produces an output vector $\boldsymbol{y}_2 \in \mathbb{R}^2$, and it is fed into the output layer, which has a single neuron with the weights $\mathbf{W}_1^{(3)}$. The output layer finally produces the prediction $\hat{\boldsymbol{y}}$. It is obvious, that each layer's output has the same dimension as the number of neurons in that layer.

All artificial neurons in the same layer usually have the same type of activation function. Let us denote the activation function in the layer i as $\phi_i : \mathbb{R}^n \to \mathbb{R}^n$, where the activation function $\phi_i(.)$ has been applied element-wise to all linear outputs of n layer neurons. If the matrix $\mathbf{W}^{(i)}$ denotes the weights of the neurons in the layer i, where each row represents the weights of one neuron, and $\boldsymbol{b}^{(i)}$ is the column vector of biases for all the neurons in the layer i, the output of the neural network can be rewritten in the form of matrix equations. Given that the output of the layer i is $\boldsymbol{y}_i$, the MLP equations for each separate layer are as follows:

$$\boldsymbol{y}_i = \phi_i(\mathbf{W}^{(i)}\boldsymbol{y}_{i-1} + \boldsymbol{b}^{(i)}) \tag{2.44}$$

Since MLP is a fully connected feedforward neural network, the functions (2.44) can be composed iteratively for adjacent layers until the output of the neural network is produced. Thus, the MLP in Figure 2.7 can be presented as the following function:

$$\hat{\boldsymbol{y}} = \phi_3\Big(\mathbf{W}^{(3)}\phi_2\big(\mathbf{W}^{(2)}\phi_1(\mathbf{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)}\big) + \boldsymbol{b}^{(3)}\Big)$$

This example is easy to extend to three hidden layers and beyond. The target of learning in MLP is to estimate the weight matrices $\mathbf{W}^{(i)}$ and bias vectors $\boldsymbol{b}^{(i)}$. The process of finding the required parameters is traditionally called learning rather than inference in machine learning [73]. Unlike the simplest types of neural networks like perceptron, MLP is able to learn complex nonlinear patterns. It has been proven by Cybenko [18] that MLP with sigmoid activation functions (2.39) and (2.40) can approximate any given continuous function with any degree of accuracy assumed that a sufficiently large number of neurons is used. This result for multilayer perceptrons is known as Universal Approximation Theorem. Later it was shown [68] that other commonly used activation functions like rectifiers (2.42) and (2.43) possess this quality too. Unfortunately, these results do not specify the estimation procedure or how the learning algorithm can discover the necessary number of neurons in MLP layers.

2.6.2 Long Short-Term Memory

Another type of neural networks, that is used for sequence processing, is recurrent neural networks (RNN). Recurrent neural networks are preferable for the data that can be represented as a sequence of inputs [55]. Figure 2.8a presents the RNN as a feedforward neural network (FNN) which feeds its output $\boldsymbol{y}_t$ back to the input. Using such loops, RNN can maintain an internal state, or memory, through time. It is also possible to view RNNs unwrapped through time as in Figure 2.8b, when the output of the FNN at time t is fed to another FNN with exactly the same structure at step $t+1$. At each t, the input is formed from the features $\boldsymbol{x}_t$ and the output of the previous FNN, $\boldsymbol{y}_{t-1}$.

Any type of the feedforward neural network can be easily converted into RNN by simply adding loops in between the layers. However, this approach usually fails for long sequences because the gradient-based methods for parameter optimization in such RNNs often either get too close to zero or too large for computer systems to process [8], which is known as the problem of vanishing and exploding gradients.

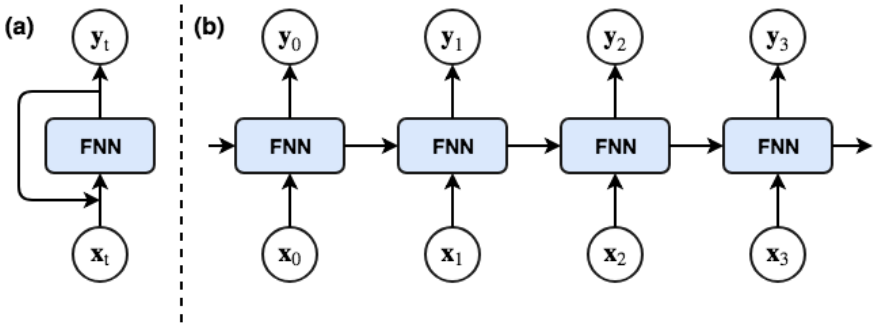

Figure 2.8: a) RNN basic view. b) Unrolled RNN.

The problems with the gradient-based optimization methods in simple RNNs prompted the new approaches to long sequence processing. Eventually a type of the RNN that solves the vanishing gradient problem, called the long short-term memory (LSTM), was introduced in 1997 by Hochreiter and Schmidhuber [45]. LSTM became popular due to its exceptional results in natural language processing [55, 78] including handwriting [40]. The schematic presentation of an LSTM layer is given in Figure 2.9.

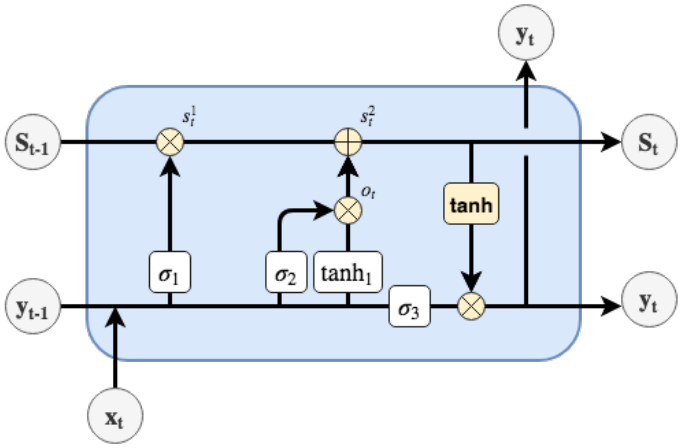

Figure 2.9: LSTM layer structure.

Suppose that the input of the LSTM is $\mathbf{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, .., \boldsymbol{x}_T\}$, $\boldsymbol{x}_t \in \mathbb{R}^n$, and the LSTM output is respectively $\mathbf{Y} = \{\boldsymbol{y}_1, \boldsymbol{y}_2, .., \boldsymbol{y}_T\}$, $\boldsymbol{y}_t \in \mathbb{R}^m$. Given the input $\boldsymbol{x}_t$ LSTM produces the output $\boldsymbol{y}_t$, and also updates its internal state (cell) which is passed to the next time step as the variable $\mathbf{S}_t$ in Figure 2.9. The upper horizontal line represents the LSTM cell, or the memory of the

network. Parallel to it, at the bottom, run both the input $\boldsymbol{x}_t$ and the output of the previous time step $\boldsymbol{y}_{t-1}$. LSTM is different from simpler types of RNNs because it has a concept of gates. The three gates are presented in the Figure 2.9 as σ_1, σ_2, and σ_3.

The initial state of the cell is received from the previous time step, $\mathbf{S}_{\text{t-1}}$. The first gate $\sigma_1(.)$, called the forget gate, takes both the previous output $\boldsymbol{y}_{t-1}$ and the new input $\boldsymbol{x}_t$, and supplies them to the function $\sigma_1 : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ which is an element-wise logistic sigmoid. The result is a vector of values from 0 to 1. The output of σ_1 is then multiplied with the previous state value $\mathbf{S}_{\text{t-1}}$ element-wise. It makes the LSTM to forget those parts of the cell, which are multiplied by the values close to 0. Given the notation in Figure 2.9, the forget gate can be defined mathematically as follows:

$$\boldsymbol{s}_t^1 = \mathbf{S}_{\text{t-1}} * \sigma_1(\boldsymbol{b}^f + \mathbf{U}^f \boldsymbol{y}_{t-1} + \mathbf{W}^f \boldsymbol{x}_t) \tag{2.45}$$

where $\boldsymbol{b}^f$ is the column vector of the bias coefficients of the forget gate, $\mathbf{W}^f$ and $\mathbf{U}^f$ are the matrices with the neuronal weights of the forget gate. The symbol $*$ denotes the element-wise multiplication.

The external input gate logistic function $\sigma_2 : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ follows after the forget-gate. The function $\tanh_1 : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ produces a new update for the sell state $\boldsymbol{s}_t^1$ based on the input vector $\boldsymbol{x}_t$ and the previous output $\boldsymbol{y}_{t-1}$ using the hyperbolic tangent. The external input gate controls which entries of this new update to keep with the element-wise sigmoid. The new update is multiplied element-wise with the output of the gate to produce the vector $\boldsymbol{o}_t$, which is added to the cell state $\boldsymbol{s}_t^1$ to produce a new cell state $\boldsymbol{s}_t^2$. This sequence of operations can be mathematically summarized as follows:

$$\boldsymbol{s}_t^2 = \boldsymbol{s}_t^1 + \sigma_2(\boldsymbol{b}^g + \mathbf{U}^g \boldsymbol{y}_{t-1} + \mathbf{W}^g \boldsymbol{x}_t) * \tanh_1(\boldsymbol{b} + \mathbf{U}\boldsymbol{y}_{t-1} + \mathbf{W}\boldsymbol{x}_t) \tag{2.46}$$

where $\boldsymbol{b}^g$ are the biases, and $\mathbf{W}^g, \mathbf{U}^g$ are the neuronal weights of the external input gate. The input transformation function $\tanh_1$ is parameterized with the vector of biases $\boldsymbol{b}$, the matrix of input weights $\mathbf{W}$, and the matrix of recurrent weights $\mathbf{U}$.

The last gate is the output gate $\sigma_3 : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$. It controls what is the output of the LSTM layer for the time step t. First, it takes the input $\boldsymbol{x}_t$, and the previous output $\boldsymbol{y}_{t-1}$, and transforms them into a vector of values from 0 to 1 using the logistic sigmoid. This vector controls which values from the cell state are output. The state $\boldsymbol{s}_t^2$ is first transformed with the element-wise hyperbolic tangent function, and then multiplied element-wise by the control sequence of the output gate. Thus, the output $\boldsymbol{y}_t$ is formed and passed to the next time step. The modified vector $\boldsymbol{s}_t^2$ becomes the new

cell state $\mathbf{S}_{\mathrm{t}}$. These equations can be written as follows:

$$\begin{aligned}\boldsymbol{y}_{\mathrm{t}} &= \tanh(\boldsymbol{s}_t^2) * \sigma_3(\boldsymbol{b}^o + \mathbf{U}^o \boldsymbol{y}_{t-1} + \mathbf{W}^o \boldsymbol{x}_t) \\ \mathbf{S}_{\mathrm{t}} &= \boldsymbol{s}_t^2\end{aligned}$$

where $\boldsymbol{b}^o$ is the vector of the output gate biases, $\mathbf{U}^o$ is the matrix of recurrent weights, and $\mathbf{W}^o$ is the matrix of input weights of the output gate. Given the gate definitions above, the complete equations for the LSTM for the time step t can be defined as follows:

$$\begin{aligned}\mathbf{S}_{\mathrm{t}} &= \mathbf{S}_{\mathrm{t\text{-}1}} * \sigma_1(\boldsymbol{b}^f + \mathbf{U}^f \mathbf{y}_{\mathrm{t\text{-}1}} + \mathbf{W}^f \boldsymbol{x}_t) + \\ &\quad \sigma_2(\boldsymbol{b}^g + \mathbf{U}^g \boldsymbol{y}_{t-1} + \mathbf{W}^g \boldsymbol{x}_t) * \tanh_1(\boldsymbol{b} + \mathbf{U}\boldsymbol{y}_{t-1} + \mathbf{W}\boldsymbol{x}_t) \qquad (2.47) \\ \boldsymbol{y}_t &= \tanh\Big(\mathbf{S}_{\mathrm{t\text{-}1}} * \sigma_1(\boldsymbol{b}^f + \mathbf{U}^f \boldsymbol{y}_{t-1} + \mathbf{W}^f \boldsymbol{x}_t) + \\ &\quad \sigma_2(\boldsymbol{b}^g + \mathbf{U}^g \boldsymbol{y}_{t-1} + \mathbf{W}^g \boldsymbol{x}_t) * \tanh_1(\boldsymbol{b} + \mathbf{U}\boldsymbol{y}_{t-1} + \mathbf{W}\boldsymbol{x}_t)\Big) * \\ &\quad \sigma_3(\boldsymbol{b}^o + \mathbf{U}^o \boldsymbol{y}_{t-1} + \mathbf{W}^o \boldsymbol{x}_t) \qquad (2.48)\end{aligned}$$

The LSTM input transform weights $\mathbf{W}$ and $\mathbf{U}$ can be viewed as sets of artificial neurons, and $\tanh_1$ can be viewed as an activation function. The number of rows in these matrices defines the number of neurons. Likewise, each gate has its own set of neurons, which are connected to other parallel clusters of neurons through non-linear operations. Thus, the LSTM can be viewed as a single layer of parallel feedforward NNs which feed their 2 non-linearly combined outputs back into their inputs.

2.6.3 Learning in Neural Networks

The MLP, LSTM and other ANN models are incomplete without a way to discover the neuronal weights. The target of learning in neural networks (NN) is to find such a combination of parameters that the neural network provides reliable and accurate predictions. Neural networks are a very broad and flexible class of models that are prone to overfitting and poor generalization when training is done incorrectly since sufficiently large neural networks have the capacity to remember the whole training set [38]. Moreover, due to the inherent complexity of many ANN functions, some traditional optimization methods are poor at their parameter estimation. Thus, there is a set of specific approaches commonly used for the training of ANN models. Most commonly and successfully used methods for parameter optimization in neural networks are based on different variations of the gradient descent, but other alternatives exist [8]. The target of learning in neural networks is to minimize the empirical risk function under the restriction of the bias-variance tradeoff phenomenon introduced earlier.

2.6.3.1 Gradient Descent and Variants

The standard way to optimize a neural network loss function involves using some optimization routine based on the gradient descent. This class of algorithms involves computing the gradient of the loss function with respect to the parameters of interest and updating the current best solution with the calculated gradient iteratively until convergence. Since the gradient points in the direction of the largest speed of increase of the function value, moving against it in a high-dimensional space implies that the function values should decrease.

Given the dataset $\mathbf{X}$ which consists of N data points $(\boldsymbol{x}_i, y_i)$, the loss function is defined as $\mathcal{L}(\mathbf{X}; \boldsymbol{\theta})$. The steepest gradient descent finds the gradient $\nabla_{\hat{\boldsymbol{\theta}}}\mathcal{L}(\mathbf{X})$ of this loss function given the data $\mathbf{X}$ with respect to $\boldsymbol{\theta}$ evaluated at $\hat{\boldsymbol{\theta}}$, and proposes the next better estimate of the function optimum. Let us define the sequence of points generated by the gradient descent as $\hat{\boldsymbol{\theta}}_t$ for $t \in (1, \infty)$. Then we can write the algorithm iteration as follows:

$$\hat{\boldsymbol{\theta}}_{t+1} = \hat{\boldsymbol{\theta}}_t - \alpha \nabla_{\hat{\boldsymbol{\theta}}_t}\mathcal{L}(\mathbf{X})$$

where the parameter α is called the learning rate, or step size, and controls how large a step should be made in the direction opposite to the gradient. By choosing a large step size, the procedure converges to the optimum faster, but has a potential to overshoot $\hat{\boldsymbol{\theta}}_{\min}$, so the algorithm might never converge. A smaller learning rate improves convergence, but is much slower.

The steepest gradient descent works with the loss function defined on the whole training set. If the gradient is computed using the loss function of 1 observation at a time, $\mathcal{L}(y_i, \boldsymbol{x}_i; \boldsymbol{\theta})$, and the algorithm picks different points at different iterations, then this type of procedure is called the stochastic gradient descent (SGD). Its update rule can be specified as follows:

$$\hat{\boldsymbol{\theta}}_{t+1} = \hat{\boldsymbol{\theta}}_t - \alpha \nabla_{\hat{\boldsymbol{\theta}}_t}\mathcal{L}(y_i, \boldsymbol{x}_i)$$

where the points $(\boldsymbol{x}_i, y_i)$ are chosen randomly without replacement from the dataset $\mathbf{X}$ one-by-one until all of them have been used at least once. One full iteration cycle through all the points in the training set is called an epoch. The SGD can be shown to converge almost surely to a local or the global minimum.

An alternative to both SGD and steepest gradient descent is the mini-batch gradient descent (MBGD). The MBGD is the most popular approach to NN optimization. The dataset $\mathbf{X}$ of N data points is partitioned into M batches randomly, and the gradient descent is computed for the loss function $\mathcal{L}$ of each batch $\mathbf{X}_m, 1 \leq m \leq M$ as follows:

$$\hat{\boldsymbol{\theta}}_{t+1} = \hat{\boldsymbol{\theta}}_t - \alpha \nabla_{\hat{\boldsymbol{\theta}}_t}\mathcal{L}(\mathbf{X}_m), \quad \forall m \in M \tag{2.49}$$

The updates (2.49) are repeated until all batches have been used once, or one epoch in MBGD. After the epoch is over, the dataset is partitioned into batches again and the whole process is repeated starting from the latest estimated parameter vector $\hat{\boldsymbol{\theta}}_t$. The epochs are repeated until sufficient convergence is achieved.

The MBGD algorithm can converge a little slower than the steepest gradient descent, and some of its steps can even diverge from the minimum, but on the average it converges to the local or global optimum. It also has an advantage that it can escape local minima, since it is only an approximation to the steepest gradient descent, and does not always move in the best direction due to the batch sample randomness. All the batches apart from the last one (it gets the leftover data points) are usually of the same size N_m, which is usually tuned along with the learning rate on the validation set. Another important consideration is the careful selection of the starting point $\hat{\boldsymbol{\theta}}_0$, which is the initial parameter guess of the algorithm. The proper setting of $\hat{\boldsymbol{\theta}}_0$ can help avoid local minima as well as make the convergence faster.

One of the hardest parts of the gradient descent based optimization is to decide what is the appropriate step size α. Very often it is too small, which results in slow decreases in objective function value, even though the movement direction may be correct. On the other hand, large values of α result in divergence. The simplest approach to this problem is to use the learning rate decay, where the step size α decreases with each epoch. But a more robust option is to use momentum [66]. This method saves the change in the parameters from the last iteration, and combines it with the scaled gradient of the current iteration. For the mini-batch gradient descent, the update with the momentum can be presented as follows:

$$\triangle_t \boldsymbol{\theta} = \beta \triangle_{t-1} \boldsymbol{\theta} - \alpha \nabla_{\hat{\boldsymbol{\theta}}} \mathcal{L}(\mathbf{X}_m) \tag{2.50}$$

$$\hat{\boldsymbol{\theta}}_{t+1} = \hat{\boldsymbol{\theta}}_t + \triangle_t \boldsymbol{\theta} \tag{2.51}$$

where the coefficient $\beta \in (0, 1)$ controls how much of the previous updates $\triangle_{t-1}\boldsymbol{\theta}$ is added to the current scaled negative gradient estimate $-\alpha \nabla_{\hat{\boldsymbol{\theta}}} \mathcal{L}$. The initial momentum value $\triangle_0 \boldsymbol{\theta}$ can be set to zero. After that, the momentum update Equation 2.50 and parameter update Equation 2.51 are iterated with different batches for many epochs until the algorithm finds a sufficiently good solution. The momentum method improves the convergence of the MBGD by preventing the gradient oscillations between batches. If the updates have had the same direction for some iterations, then the momentum is large, and the new update will partially disregard the gradient direction change if the new gradient is different from the previous gradients.

Another way to improve the MBGD convergence is to use the root mean

square propagation (RMSProp). This technique uses the scaling of the gradient descent learning rate for each updated parameter separately [65]:

$$\boldsymbol{v}_t = \gamma \boldsymbol{v}_{t-1} + (1-\gamma)\big(\nabla_{\hat{\boldsymbol{\theta}}} \mathcal{L}(\mathbf{X}_m)\big)^2 \tag{2.52}$$

$$\hat{\boldsymbol{\theta}}_{t+1} = \hat{\boldsymbol{\theta}}_t - \frac{\alpha}{\sqrt{\boldsymbol{v}_t + \boldsymbol{\epsilon}}} * \nabla_{\hat{\boldsymbol{\theta}}} \mathcal{L}(\mathbf{X}_m) \tag{2.53}$$

where $\gamma \in (0, 1)$ is a parameter that controls how much the vector of second moments $\boldsymbol{v}_t$ should be updated per each iteration of the gradient descent. The square of the gradient in Equation 2.52 is the element-wise operation, the asterisk operator $*$ denotes the element-wise multiplication, $\sqrt{\boldsymbol{v} + \boldsymbol{\epsilon}}$ is the element-wise square root, and the division is also element-wise in Equation 2.53. The vector $\boldsymbol{\epsilon}$ is filled with small values which ensure that the denominator does not become zero. The RMSProp algorithm allows to shrink the large component updates in the gradient descent and prevent oscillations around the best minimization direction when the mini-batches are used. The initial value $\boldsymbol{v}_0$ can be set to zeros, and then Equations 2.52 - 2.53 are iterated until all batches have been used once. The epochs are iterated until convergence.

2.6.3.2 Backpropagation

The computation of the gradients in neural networks is nontrivial, since ANNs have many layers with their own sets of parameters, and the outputs of layers are combined with non-linear activation functions to produce the output. The method used to compute the gradient in ANNs is called backpropagation. The backpropagation recursively computes the gradients for each layer in the neural network using the chain rule. It gets its name from the fact that it uses the errors from the deeper (closer to the output) layers to compute the gradients for the preceding layers, which can be seen as error propagation in the direction opposite to the data flow. The backpropagation version for the RNNs is called the backpropagation through time, since it unrolls the RNN as in Figure 2.8b, and propagates the error through the existing layers, as well as through the past steps of the RNN input.

The backpropagation algorithm can be illustrated by using the feedforward fully-connected NNs. Let us use the layer view of the MLP from Equation 2.44. We further introduce the linear response of the layer i as $\boldsymbol{a}_i$. The input-output equations of the layer i are as follows:

$$\boldsymbol{a}_i = \mathbf{W}^{(i)} \boldsymbol{y}_{i-1} + \boldsymbol{b}^{(i)}$$
$$\boldsymbol{y}_i = \phi_i(\boldsymbol{a}_i)$$

where $\mathbf{W}^{(i)}$ and $\boldsymbol{b}^{(i)}$ stand for the weights and biases of the neurons in the layer i, while $\phi_i(.)$ is the activation function of this layer. This can be viewed as a transformation from the input $\boldsymbol{y}_{i-1}$ to the output $\boldsymbol{y}_i$. The input, output, and linear response of a layer are all regarded as the variables that depend on the previous layers' parameters in the backpropagation routine.

Given an MLP neural network with n layers, assume the loss function $\mathcal{L}(y, \hat{\boldsymbol{y}})$, where y is the label, and $\hat{\boldsymbol{y}}$ is the output of the MLP neural network for the input features $\boldsymbol{x}$ such that $\hat{\boldsymbol{y}} = \boldsymbol{y}_n = \phi_n(\boldsymbol{a}_n)$. Let us introduce the error vector as follows:

$$\boldsymbol{e}_i = \frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \boldsymbol{y}_i} \tag{2.54}$$

where $\boldsymbol{y}_i$ is the output of layer i. Now, we can use the chain rule to find the derivatives of the loss function with respect to any parameters of the neural network by recursively computing the error terms from the deepest layer n to the first layer. The chain rule looks as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \mathbf{W}^{(i)}} &= \frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \boldsymbol{y}_i} \times \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{a}_i} \times \frac{\partial \boldsymbol{a}_i}{\partial \mathbf{W}^{(i)}} \\
\frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \mathbf{b}^{(i)}} &= \frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \boldsymbol{y}_i} \times \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{a}_i} \times \frac{\partial \boldsymbol{a}_i}{\partial \boldsymbol{b}^{(i)}} \\
\boldsymbol{e}_i &= \frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \boldsymbol{y}_{i+1}} \times \frac{\partial \boldsymbol{y}_{i+1}}{\partial \boldsymbol{a}_{i+1}} \times \frac{\partial \boldsymbol{a}_{i+1}}{\partial \boldsymbol{y}_i}
\end{aligned} \tag{2.55}$$

where Equation 2.55 suggests that the error in the layer i can be computed recursively from the error in layer $i+1$, given the notation of Equation 2.54. All the derivatives are with respect to a vector, or a matrix. We introduce the partial derivatives of the layer's response $\boldsymbol{y}_i$ with respect to its linear response $\boldsymbol{a}_i$ as a matrix $\Phi^{(i)}$. This matrix is diagonal and its dimensions equal the number of neurons in the layer i. Its diagonal entries are $\Phi^{(i)}_{jj} = \phi'_i(\boldsymbol{a}_{ij})$ where $\phi'_i(.)$ is the derivative of the activation function in the layer i with respect to its argument, and $\boldsymbol{a}_{ij}$ is the linear response of the neuron j in the layer i. Now the relevant partial derivatives of the MLP loss function can be stated formally as follows:

$$\boldsymbol{e}_i = \mathbf{W}^{(i)T} \Phi^{(i)} \boldsymbol{e}_{i+1} \tag{2.56}$$

$$\frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \mathbf{W}^{(i)}} = \Phi^{(i)} \boldsymbol{e}_i \boldsymbol{y}_{i-1}^T \tag{2.57}$$

$$\frac{\partial \mathcal{L}(y, \hat{\boldsymbol{y}})}{\partial \boldsymbol{b}^{(i)}} = \Phi^{(i)} \boldsymbol{e}_i \tag{2.58}$$

where the errors are column vectors.

In the neural network with n layers, the backpropagation starts by initializing the error term $\boldsymbol{e}_n = \frac{\partial}{\partial \hat{\boldsymbol{y}}}\mathcal{L}(y, \hat{\boldsymbol{y}})$. Then, for each layer from n to 1, the backpropagation computes the errors recursively with Equation 2.56. The weight and bias partial gradients are computed from Equations 2.57 and 2.58 as soon as the error $\boldsymbol{e}_i$ is available. After the parts of the gradient have been computed through all the layers, the weight and bias parameters can be updated in one gradient descent step:

$$\mathbf{W}_{t+1}^{(i)} = \mathbf{W}_t^{(i)} - \alpha \nabla_{\mathbf{W}^{(i)}} \mathcal{L}(y, \hat{\boldsymbol{y}}) \tag{2.59}$$

$$\boldsymbol{b}_{t+1}^{(i)} = \boldsymbol{b}_t^{(i)} - \alpha \nabla_{\boldsymbol{b}^{(i)}} \mathcal{L}(y, \hat{\boldsymbol{y}}) \tag{2.60}$$

where α is the learning rate of the gradient descent algorithm.

The backpropagation in recurrent neural networks including LSTM follows the same principles (chain rule, recursion) as the backpropagation in MLP. However, the errors in the RNN are also propagated through time, from the end towards the beginning of the input sequence. Complex backpropagation routines like those in RNNs are usually executed through computational graphs in various libraries like Tensorflow [1]. The backpropagation is a very flexible procedure. Due to the chain rule, very complex neural network function gradients can be decomposed in terms of sums, products, and well-known activation function derivative forms. This allows to create rich representations of data using ANNs.

2.6.3.3 Regularization

There is a broad range of techniques to prevent overfitting in neural networks. The simplest regularization approaches in ANNs are based on L_2 and L_1 norm penalties discussed in Section 2.5.3. Given the loss function $\mathcal{L}(y, f(\boldsymbol{x}; \boldsymbol{\theta}))$, where y is the label, $\boldsymbol{x}$ is the vector of features, and $f(.; \boldsymbol{\theta})$ is a neural network function with the vector of parameters $\boldsymbol{\theta}$ of size N, the L_2 and L_1 regularized loss can be presented as follows:

$$\mathcal{L}_{\mathrm{L}_2}(y, f(\boldsymbol{x}; \boldsymbol{\theta})) = \mathcal{L}(y, f(\boldsymbol{x}; \boldsymbol{\theta})) + \frac{\lambda}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}$$

$$\mathcal{L}_{\mathrm{L}_1}(y, f(\boldsymbol{x}; \boldsymbol{\theta})) = \mathcal{L}(y, f(\boldsymbol{x}; \boldsymbol{\theta})) + \frac{\lambda}{2}\sum_{i=1}^{N}|\boldsymbol{\theta}_i|$$

where the parameter λ controls the strength of the regularization. The vector of parameters $\boldsymbol{\theta}$ can include either all parameters of the neural network, or just some of them. However, the most common approach is to regularize only the neuronal weights $\mathbf{W}^{(i)}$. The L_1- and L_2-norm regularization can be applied with different strength to separate layers and neurons [38].

Another regularization technique which has become recently popular is dropout. The technique prevents overfitting by dropping the units of the neural network randomly during training, reducing its capacity to adapt to the data too much [69]. It can be applied to the layers of the neural network separately. In the case of LSTM, the dropout can be used with different gates independently. The regularization strength of this method is controlled with the parameter $p_i \in (0, 1)$, which specifies the probability of a single unit in the layer i to be switched off during one iteration of the gradient descent. Given the output vector $\boldsymbol{y}_i$ of the layer i as defined earlier in (2.44), we can specify the mask $\boldsymbol{d}_i$ of the same dimension as $\boldsymbol{y}_i$, such that each element of the vector $\boldsymbol{d}_i$ is either 1 with probability p_i, or 0 otherwise. When the neural network processes the input feature vector, the output $\boldsymbol{y}_i$ of the dropout layer i is transformed with the following mapping

$$\boldsymbol{y}_i^{\text{dropout}} = \frac{1}{p_i}\boldsymbol{d}_i * \boldsymbol{y}_i \tag{2.61}$$

During the backpropagation, the vectors $\boldsymbol{y}_i^{\text{dropout}}$ are used instead of $\boldsymbol{y}_i$. The mapping (2.61) does not happen during the prediction.

A very popular regularization technique is called early stopping [38]. This method relies on the preliminary stopping of the optimization algorithm such as the gradient descent before it finds the minimum of the loss function on the training set as soon as the validation set accuracy metric starts showing that the neural network begins overfitting as demonstrated in Figure 2.10.

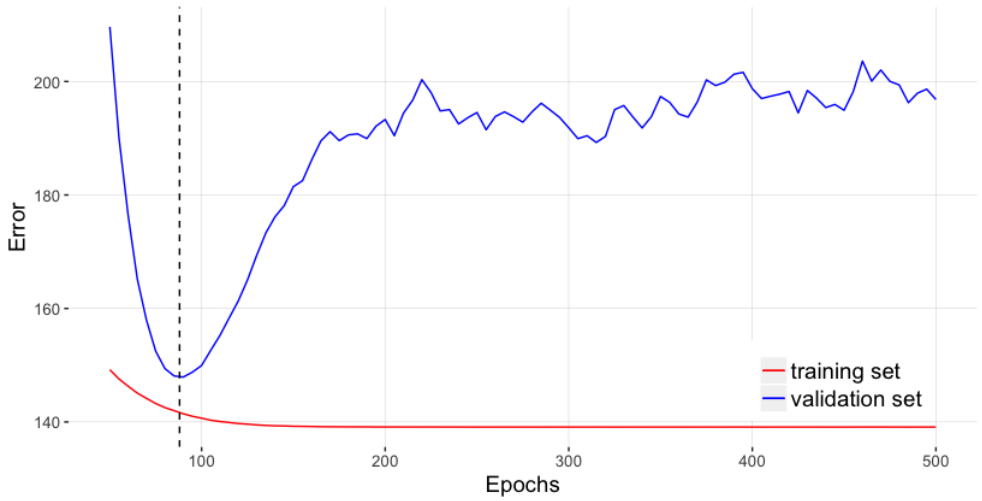

Figure 2.10: Early stopping point for neural network regularization.

In Figure 2.10 it is seen that the validation set error starts rapidly growing around 90 epochs into the training, which is indicated by the dashed vertical line. At the same time the training set error continues to decrease. However, the validation set error is the indicator of how well the neural network fits the out-of-sample data points, and is the definitive indicator of the generalization ability of the ANN. The early stopping regularization routine checks the validation set error regularly, and when the error on the validation set stops decreasing for a few epochs, the current parameters are selected as the final ones. In the figure, the set of the best parameters is found around the 90th epoch, and the optimization procedure is terminated.

2.6.3.4 Batch Normalization

We can view each hidden layer in the neural network as a function that transforms its input $\boldsymbol{y}_{i-1}$ from the previous layer into the output $\boldsymbol{y}_i$ according to Equation 2.44. During the mini-batch gradient descent optimization, neuronal weights of hidden layers change after each MBGD iteration. As a result, the NN hidden layers produce varying outputs at different time steps for the same inputs. It indicates, that the distribution of the hidden layer inputs changes during the mini-batch gradient descent. This phenomenon is known as the internal covariance shift. It necessitates the use of smaller learning rates in deep neural networks, makes the careful parameter initialization very important, and reduces the effectiveness of saturating activation functions [50].

In order to avoid the problems caused by the covariance shift, a procedure called **batch normalization** is introduced in deep neural networks. It is based on the hidden layer input normalization for each batch in MBGD, and allows to use much higher learning rates, and simpler weight initialization techniques [50]. The batch normalization also acts as a weak regularizer.

Assume the full training set size is N. Suppose that we have a mini-batch of size m, and the respective activation outputs for the layer i, which are denoted as $\{\boldsymbol{y}_1^{(i)}, .., \boldsymbol{y}_m^{(i)}\}$. The batch normalization function $\zeta_{\boldsymbol{\gamma}_i, \boldsymbol{\beta}_i}$ maps these outputs as

$$\zeta_{\boldsymbol{\gamma}_i, \boldsymbol{\beta}_i} : \boldsymbol{y}_j^{(i)} \rightarrow \boldsymbol{z}_j^{(i)}, \quad \lim_{m \rightarrow N} \boldsymbol{z}_j^{(i)} = \boldsymbol{y}_j^{(i)}$$

where the subscript notation suggests that the function ζ depends on 2 vector parameters: $\boldsymbol{\gamma}_i$ and $\boldsymbol{\beta}_i$. These two parameter vectors are estimated during the backpropagation in addition to normal neuronal weights and biases. The function ζ is in fact an identity transform [50]. Its purpose is to learn the parameters $\boldsymbol{\gamma}_i$ and $\boldsymbol{\beta}_i$, which represent the full training sample mean and

variance for the inputs in layer i, that can get distorted due to mini-batches. Even if the mini-batch sample mean or variance get significantly distorted during training, the batch normalization transform will normalize the layer input back to its full sample mean and variance with $\boldsymbol{\gamma}_i$ and $\boldsymbol{\beta}_i$. The full form of the batch normalizing function is as follows [50]:

$$\zeta_{\boldsymbol{\gamma}_i,\boldsymbol{\beta}_i}(\boldsymbol{y}_j^{(i)}) = \boldsymbol{\gamma}_i \left(\frac{\boldsymbol{y}_j^{(i)} - \mathrm{E}\big[\boldsymbol{y}_j^{(i)}\big]}{\sqrt{\mathrm{Var}\big(\boldsymbol{y}_j^{(i)}\big) + \boldsymbol{\epsilon}}} \right) + \boldsymbol{\beta}_i \tag{2.62}$$

where the constant $\boldsymbol{\epsilon}$ is a vector of small numbers to prevent the division by zero. The multiplication, division, and square root operations are element-wise. The sample expectation and variance are computed with the sample of outputs $\boldsymbol{y}_j^{(i)}, 1 \leq j \leq m$ corresponding to the currently processed mini-batch. Both statistics are vectors: a vector of expectations, and a vector of variances (not the covariance matrix). The variance estimator is biased, and computed by dividing with m instead of $m-1$ [50]. The output of the batch normalization function for the layer i is a vector.

The function ζ is differentiable, and can be applied to all or some of the hidden layers in the deep neural network. The parameters $\boldsymbol{\gamma}_i$ and $\boldsymbol{\beta}_i$ (one separate pair of parameter vectors for each layer where the batch normalization is applied) are learned with the gradient descent algorithm. After the training, we calculate the statistics $\mathrm{E}\big[\boldsymbol{y}^{(i)}\big]$ and $\mathrm{Var}\big(\boldsymbol{y}^{(i)}\big)$ using the whole training set opposite to using mini-batches before. Then, we can make the predictions by applying the function $\zeta_{\boldsymbol{\gamma}_i,\boldsymbol{\beta}_i}$ with the learned parameters $\boldsymbol{\gamma}_i$, $\boldsymbol{\beta}_i$ and sample statistics computed on the whole training set to the activation outputs in the layers with the batch normalization. Since the batch normalization parameters are estimated to represent an identity function given the full training set, the transformation should not distort the layer outputs during prediction. Using the batch normalization mappings allows to approximate the underlying distribution of the inputs in each layer during training. Even though the batch normalization statistics $\mathrm{E}\big[.\big]$ and $\mathrm{Var}\big(.\big)$ can change a lot during the MBGD depending on a given batch, the discovered parameters $\boldsymbol{\gamma}_i$ and $\boldsymbol{\beta}_i$ should correct for this random variation.

Chapter 3

Related Work

Demand forecasting practices in various types of businesses have significantly evolved in the past 50 years, specifically due to introduction of new statistical methods and software. The earliest forecasting practices in supply chains were heavily based on judgmental forecasting and basic estimation procedures such as moving averages, simple exponential smoothing and naive methods which was indicated by many surveys held in the US and the UK [20, 26, 67]. It was found, that the prevalence of specific techniques depended on the type of the organization and the size of the company producing the forecasts. Organizations of all sizes used judgmental techniques equally often, but more sophisticated quantitative methods were usually employed at larger companies, while smaller companies relied more on simpler quantitative extrapolation methods [67]. It slowly became obvious that judgmental forecasting had a lot of drawbacks, and forecasting started moving towards more advanced quantitative procedures. In the past 50 years, a broad range of quantitative methods have been used to predict demand such as various types of regression and state-space modelling, exponential smoothing, autoregressive integrated moving average (ARIMA), and various machine learning techniques including neural networks (NN).

A significant amount of research into predictive modelling originates outside of grocery retail. Comparative analyses of demand forecasting models are usually conducted for large manufacturers, utility providers (gas, water or electricity), and tourism businesses, while similar research is scarce for shops. However, stochastic processes that drive sales in retail and other industries, such as tourism business, or manufacturing [35], are often similar, so the set of models used to predict demand outside of retail is often applied to demand forecasting in grocery retail, even though some attention should be paid to the frequency of historical data and forecasts, level of forecast automation, and specific demand patterns in different studies.

ARIMA and exponential smoothing with trend and seasonality are considered to be very popular models for demand forecasting. ARIMA models, that have a rigorous theoretical background, were popularised by Box and Jenkins in 1970s [48, 71]. Since then, they have been used in many comparative studies of demand forecasting, recently against neural networks. However, in literature review by Syntetos et al. [71], exponential smoothing techniques are stated to be the most frequently used forecasting methods in supply chains. Miller et al. [58] tests a wide range of Holt-Winters (HW) exponential smoothing methods on the aggregated sales of 28 product families of a process manufacturer. In this research, a vast superiority of exponential smoothing methods over naive forecasting is shown. In a study of department store retail sales, Geurts and Kelly [35] come to a conclusion that exponential smoothing outperforms both judgmental forecasting, as well as ARIMA models. Gamberini et al. [32] compares additive seasonal HW exponential smoothing model to seasonal ARIMA (SARIMA) with respect to predicting intermittent demand for different forecast horizons. HW compares favourably to SARIMA models for shorter forecast horizons, but seasonal ARIMA seems to be universally better when the trend and seasonal components remain consistent in the time series for longer horizons.

Arunraj and Ahrens [6] present a hybrid of seasonal autoregressive integrated moving average with external variables (SARIMAX) and quantile regression, called SARIMA-QR. They compare it to naive forecasting, multilayer perceptron (MLP) with 2 different architectures, plain SARIMA, and SARIMAX. The authors use a daily time series of banana sales from a food retailer to analyze the accuracy of demand forecasts. The dataset has a large number of covariates, such as weather recordings, holiday and promotion indicators, and month indicators. The authors use mean absolute percentage error (MAPE) and root mean squared error (RMSE) to evaluate the final accuracy of the models on the test set. While both SARIMAX and SARIMA-QR are better than neural networks in MAPE, multiple hidden layer MLP architecture is better than both SARIMA-based models in RMSE. Arunraj and Ahrens note, that SARIMA-QR is a more valuable model in demand forecasting, since it allows to estimate confidence bounds for future sales without specific error distribution assumptions, which helps the management to make decisions when they are interested in the worst or the best scenarios of the future demand.

With the recent advent of neural networks and deep learning, a lot of recent research has concentrated on neural networks and their applications to demand forecasting. Alon et al. [4] compares feedforward neural networks to more traditional models such as Holt-Winters, Box-Jenkins ARIMA, and multivariate regression. The model performance is assessed for the monthly

aggregate retail sales time series provided by the U.S. Department of Commerce. In the study, there are 2 test sets for different time periods, one of which is marked by macroeconomic instability, that directly affects aggregate retail sales by making them fluctuate. Forecasts have different horizons, up to 12 months into the future, and the performance metric is MAPE. The artificial neural networks (ANN) provided the best forecasts for all horizons for the period of macroeconomic instability, followed by ARIMA, while for the second test set with fewer fluctuations, ARIMA turned out to be the best model for the one-step forecast, and exponential smoothing created the best multi-step forecasts. The authors conclude that ANN as a whole performed the best across both test sets combined, while showing superior accuracy in the periods of economic volatility. They make an interesting observation that for such periods of fluctuations, multi-step forecasts are paradoxically better than one-step forward forecasts.

Chu and Zhang [15] compare linear models (SARIMA and linear regression) against feedforward neural networks with different feature representations (dummy variables against trigonometric functions to model seasonality) in forecasting monthly retails sales compiled by the US Bureau of the Census. Models of each class are selected with cross-validation, while out-of-sample performance of all final models is measured on a holdout dataset. Chu and Zhang identify three-layer feedforward neural network with deseasonalized time series to be the best method for aggregate sales forecasting. The authors arrive at a conclusion that nonlinear models in general are preferable to linear ones for demand forecasting in retail. Their other interesting finding, that contradicts previous results [76], is that trigonometric modelling of seasonality results in inferior accuracy of retail sales time series forecasting. In another work, Zhang and Qi [81] assess the performance of neural networks against SARIMA on monthly retail sales as well as artificially generated time series, which have multiple seasonalities and the trend. The authors select the best seasonal ARIMA model using in-sample data, while the best NN model is selected using cross-validation. Additionally, Zhang and Qi estimate the performance of neural networks on the original time series in contrast with deseasonalized or detrended time series. They find, that three-layer feedforward NN model with deseasonalized and detrended time series data outperforms SARIMA in demand forecasting of all retail time series across all performance metrics (RMSE, MAE and MAPE). The authors find, that in cases when the data is not preprocessed properly, neural networks are worse that ARIMA models in forecasting demand. Consequently, Zhang and Qi conclude that data preprocessing is critical for neural network performance.

An interesting approach to retail sales forecasting is suggested by Aburto

and Weber [2]. The authors present a novel approach to short-term demand forecasting in a Chilean supermarket, which involves a hybrid of SARIMA with explanatory variables (SARIMAX) and a multilayer perceptron neural network, which fits the errors of the SARIMAX process. The time series in the study are presented on a daily level and have a large number of explanatory variables, most of which denote specific events such as holidays, or the price of the forecast product. The external regressors represent the features of both SARIMAX and the neural network, while NN also accepts autoregressive terms. Aburto and Weber find that neural network models with any window size outperform ARIMA models in case they are used separately, while the suggested hybrid model has the lowest test errors (MAPE, NMSE) out of all the models. All forecasting methods in the study outperform the naive and seasonal naive baseline forecasts.

While a lot of research in neural networks applied to retail sales prediction focuses on multilayer feedforward neural networks with one hidden layer, recently new studies about retail demand forecasting featuring deep learning and recurrent neural networks have emerged. Flunkert et al. [27] proposes a novel recurrent neural network (RNN) architecture named DeepAR, and compares it to the state-of-the-art forecasting methods for intermittent demand as well as simpler versions of recurrent neural networks on a variety of time series, including weekly retail sales data from Amazon, which contain both fast and slow-moving products, as well as new product introductions. The suggested model has the encoder-decoder long short-term memory (LSTM) architecture (sequence-to-sequence learning), and produces probabilistic forecasts based on a variety of possible error distributions. The dataset includes multiple features apart from sales, that can include original explanatory variables or dummies responsible for the week number, time series offset, or item category. According to Flunkert et al., their approach allows to model group-dependent dynamics with minimum effort, and produce forecasts for the products with no history since it uses all available product sales histories for training. The authors present the empirical evidence that DeepAR outperforms other alternatives in standard forecasting error measures (normalized deviation and normalized RMSE) as well as quantile error measures. Another interesting finding is that their model requires little tuning, which is usually not the case for neural network models.

Wen et al. [75] proposes a novel neural network architecture, called MQ-RNN, based on sequence-to-sequence learning, and multistep quantile forecasts. MQ-RNN is an encoder-decoder type algorithm, where the encoder is a typical RNN such as LSTM, while its decoder consists of two stacked MLP layers (global and local branches), where the global branch works with all the future inputs and RNN output at the same time, and passes horizon

specific contexts as outputs to the local branch, where the next MLP outputs the forecast quantiles. The authors argue that this two-layered architecture allows MQ-RNN to better model seasonalities and future events, and overcomes the problem of accumulating error in multistep forecasts. The error is computed as the quantile loss, used in quantile regression. MQ-RNN trains on a dataset where each training sample is one full time series with local and global (product category, identity, or other constant features) covariates. The authors argue that even though constant features do not influence individual forecasts as much as local features, they allow to share temporal information among time series, and generate forecasts for stock keeping units (SKUs) with few or no sales. During training, the decoder makes a forecast, computes and backpropagates the error for each step in the time series. Wen et al. apply their approach to demand forecasting at Amazon for 60,000 products from different categories. The authors compare MQ-RNN against itself with minimal variants, where they switch off some functionalities of their method to partly mimic other state-of-the-art algorithms, such as LSTM or DeepAR. The authors report that their method gives the best accuracy and confidence bounds among its simplified variants.

Other machine learning methods outside neural networks have been reported in several retail demand forecasting studies. Hansen et al. [42] compares support vector regression to multiple variations of ARIMA with normal and non-normal error assumptions. The forecasts are evaluated for a variety of economic time series including retail. As a result, support vector regression marginally outperforms traditional statistical methods in most forecasting tasks including retail demand forecasting. The authors point to the excellent ability of support vector regression to model nonlinear trend-cycles and seasonalities, but mention the lack of studies about time series characteristics that make the support vector regression a preferable method. Ali et al. [3] evaluates stepwise linear regression, support vector regression with three different kernels, and regression trees in grocery store demand forecasting on a weekly level in the presence of promotion information. Exponential smoothing is used as a benchmark, as well as for feature engineering. The authors use multiple data representations where each product-location week sales and its regressors (both real and artificially generated) are data points, and distinguish between different product categories, stores and SKUs with the help of dummy variables. Ali finds that regression trees have the best accuracy at the expense of the data preparation costs and algorithm complexity. Another interesting finding is that machine learning techniques significantly improve the forecasting accuracy during the promotion weeks, while they do not have any benefit in comparison to the exponential smoothing baseline during the weeks with no promotions.

Chapter 4

Methods

The target of this thesis is to find the best model class for demand forecasting of the fast-moving products. The forecasts themselves are on a daily level, have a horizon of 7 days, and should take external weather variables and promotion information into consideration. The most promising models identified from the literature so far include the ARIMA family of models and neural networks. These two classes of models are either tested or mentioned in some way in most research literature dedicated to the retail demand forecasting as per Chapter 3. However, these models have never been adequately compared in terms of multistep forecasts on a daily level in a larger subset of fast-moving products from the grocery retail. Given the results achieved with these forecasting methods for intermittent demand, as well as demand forecasting on a weekly and monthly level, it is probable that these models are most suitable for the problem of multistep daily forecasts for fast-moving products in grocery retail.

Even though various exponential smoothing models like Holt-Winters (HW), also seem to be promising in sales forecasting, these models are usually applied to univariate time series, while the sales data in our investigation involves multiple covariates in addition to sales themselves. There are some attempts in the literature to overcome this obstacle. For example, Pfeffermann and Allan [62] propose a multivariate extension of the HW exponential smoothing and apply it to the multivariate time series of hotel room demand. The work shows that the proposed extension of the HW model is better than its univariate counterpart, and is comparable to multivariate ARIMA for short forecast horizons. Another drawback of exponential smoothing models is that they lack a capability to deal with multiple seasonality, which is also present in our experimental dataset. Taylor [72] reviews the capabilities of exponential smoothing in the setting of two seasonalities and introduces double seasonal Holt-Winters model which fares well in comparison to ARIMA

models in the prediction of electricity demand. Nevertheless, the author notes that the double seasonal Holt-Winters method should be combined with other forecasting techniques for better results. Thus, various extensions of exponential smoothing that deal with regressors and multiple seasonality are uncommon, have many limitations, and the scarce applied research makes us doubt their efficacy in multivariate multiseasonal multistep demand forecasting.

The original ARIMA formulation cannot deal with external regressors. Thus, we use a regression model for weather and campaign status covariates, while using the ARIMA to model the regression errors. Multiple seasons are easily modelled with the trigonometric functions of sines and cosines and passed on to the regression with the other features. This approach is called the **dynamic harmonic regression** [48]. This model is compared against two types of deep neural networks: MLP and LSTM. The MLP is one of the most commonly used non-linear algorithms in time series analysis which is evident by the amount of research using the modifications of this model. It can work with multivariate, non-stationary time series, and multiple seasonalities. At the same time, LSTM has not been as common in demand forecasting, but it has been recently gaining popularity due to its exceptional results in other fields. Since the LSTM is specifically tailored to work with sequential data, while it also has the same benefits as the MLP in time series forecasting, it can potentially produce excellent results in fast-moving product demand forecasting.

Since many time series in the dataset are count data, Poisson GLM is also added to the compared model set. It is a well-known generalized linear model, which is simple and gives good results in many regression tasks that involve positive countable response variables. It is extended to handle multiple seasonality through trigonometric feature engineering as in dynamic harmonic regression case. Since overfitting is also a concern, the Poisson GLM is equipped with an elastic net to control for the model complexity.

All the models are compared against each other on a big dataset of long sales histories. All the models are compared against the naive seasonal forecasting, which is simple to implement and commonly used as a baseline. The tuning and final evaluation of the models is done with cross-validation (CV). The cross-validation is preferred to other methods of model selection in order to standardize the methodology across all model classes. In evaluation, the cross-validation is the only procedure that can reliably assess the out-of-sample performance of all presented models unlike AIC or other information criteria as mentioned in Sections 2.2.2 and 2.2.4. The details of the feature engineering, model selection and evaluation, as well as the theory and practical use of the compared models are discussed further in full detail.

4.1 Time Series Selection and Properties

The experimental data is carefully selected to represent the fast-moving product demand. A collection of real sales histories is received from one of European supermarket chains, and contains information about tens of thousands of products sold in 13 different locations. The sales data is aggregated on a daily level, and all records are non-negative. Additional information includes promotions for product-locations (PL), and weather data. A subset of time series is further selected according to the following criteria:

1. Length of more than 4 years of daily data
2. At least 10000 sales overall
3. The number of days that have no sales is less than 3%
4. Each time series has only 1 type of promotions
5. The first promotion always happens during the first year of sales history

The first three criteria ensure that the selected histories represent fast-selling products. The last two criteria are introduced to make the training and test sets represent the same distribution, since different campaign types can produce various changes in demand. The sales histories with a single type of campaign, which is represented in the training, validation and test sets, do not have this problem. The final dataset is formed from 100 product-location time series that are selected randomly so that each location has no more than 9 product sales histories.

In the final dataset, most time series have the length of 1826 days. 68% time series have only weekly seasonality, 29% have both yearly and weekly seasonalities, 2% have a half-yearly seasonality and a week seasonality, while only 1 product-location has a sales history with all three seasonalities. 87% of the final sales histories are count time series, while 13% are represented by fractional numbers. Figure 4.1 presents the sample means of the time series in the dataset. The figure shows that the majority of product-locations (> 50) are sold around 2 to 23 items per day. However, some PLs have large average sales up to 53 items/units per day.

Figure 4.2 presents the maximum sales against the 90th percentile of sales per day for each product-location time series in the dataset. The graph reveals that while 90 percent of all days for all product-locations have at most 125 sales, there are some days for many PLs when the sales are significantly higher than usual. Many time series in the dataset are heterogeneous, with possible outliers and level shifts.

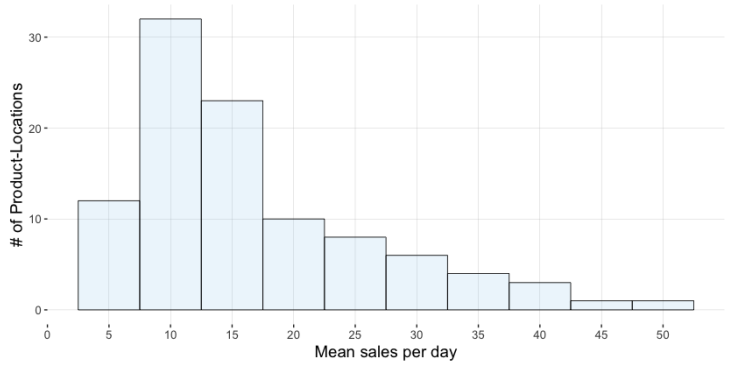

Figure 4.1: Mean sales per day for each product-location.

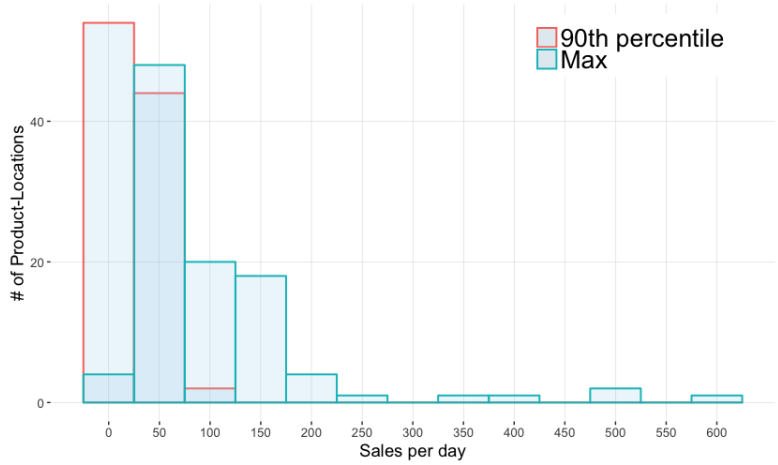

Figure 4.2: Maximum sales versus 90th sample percentile of sales per day.

The sales histories of product-locations are accompanied by covariates for each day of sales. The regressors include the campaign status binary variable, which is 1 if the sales of a given PL happened during the campaign, or 0 otherwise. Additionally, there are the temperature and wind speed records for each day of the sales history data for all product-locations. These regressors are floating points numbers. Temperature ranges from $-2.23°C$ to $30.58°C$, while the wind speed ranges from 0.01 to 15.7 m/s.

4.2 Model Selection and Evaluation

In the final dataset of 100 chosen product-locations, each time series is further split into three consecutive parts which are called the training, validation and test parts. These parts are not the same as the training, validation and test sets discussed in Chapter 2: the time series parts serve as the boundaries for the time series cross-validation procedures. The training part of the time series contains most of the observations from the beginning of the time series and is always used for training the models. It is followed by the validation part which includes the folds of the CV procedure for model (hyperparameter) selection. The last segment of the time series is the test part. It contains the consecutive folds of the CV used for the final model out-of-sample performance assessment and reporting of the experimental results. This time series division is schematically presented in Figure 4.3.

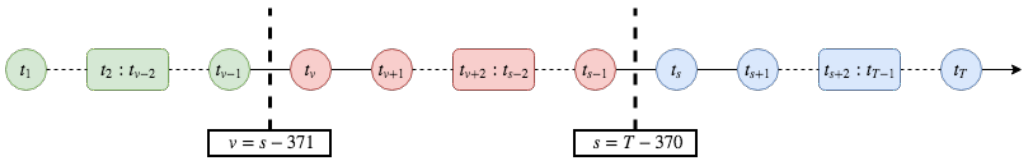

Figure 4.3: Time series division into train, validation and test parts.

Each time series is represented as a separate entity with its own train, validation and test parts. The last $371 \times 2 = 742$ days of each time series are selected for the validation and test parts. These two consecutive parts are of equal size of 371 days, or 53 weeks. The validation and test parts span the whole year each in order to include all the seasons and holidays into the out-of-sample performance measurement. The rest of the sales history is always used for training in cross-validation. In Figure 4.3, the test part of the time series with T observations starts at t_s and is colored blue. The red validation part preceding it spans from t_v to t_{s-1}. The training part is colored green, starts at t_1 and continues up to t_{v-1}.

4.2.1 Time Series Cross-Validation

All compared models have hyperparameters to be selected. As described in Chapter 2, one traditional way to compare models on the cross-sectional data is cross-validation. However, temporal dependencies in the time series make the direct application of CV impossible because its estimate of out-of-sample performance will be biased in the presence of correlated errors [9]. A

specific modification of the cross-validation for time series is known as the **evaluation on a rolling forecasting origin** [48]. Given the test fold, this procedure uses only the preceding part of the time series for training. This way the time dependency of observations can be properly preserved. After each fold has been evaluated, the final result is obtained by averaging. The procedure is illustrated schematically in Figure 4.4.

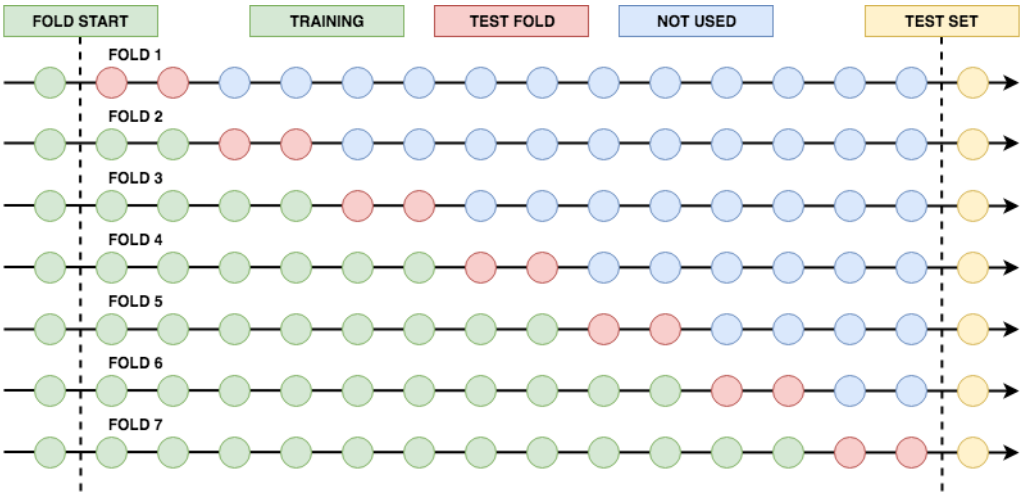

Figure 4.4: Time series 2-step forward rolling origin cross-validation.

Figure 4.4 presents an example of time series cross-validation for a 2-day forward forecasting model hyperparameter selection. In the figure, each line represents one iteration (fold evaluation) of the CV procedure. The CV folds are colored red, while the training portion of the time series is colored green. The test folds are evaluated in the order of the time series. The blue color denotes the time steps of the time series which follow the currently evaluated fold, and are not used until the fold that contains them is reached. After the prediction is obtained for the fold i, it is recorded and the next fold is fetched by moving two time steps forward. As the cross-validation proceeds, the training part of the time series grows, so that it always ends right before the evaluated fold. The CV folds for model selection cover the range of the time series (denoted with the dashed lines in Figure 4.4) from the start of the validation part of the time series until the test part of the time series. The cross-validation procedure for the model evaluation is similar to the one for hyperparameter selection, but its folds cover the test part of the time series. During the model evaluation, both the training and validation parts serve as the training data in each CV iteration.

As in the example above, the cross-validation used in the experimental part involves two stages:

1. Cross-validation with folds specified on the validation part of the sales time series, which is used to find the optimal model hyperparameters.

2. Cross-validation with folds specified on the test part of the sales time series, that is used to give the final estimate of the model performance.

The repeated cross-validation procedure allows to avoid the overfitting which can arise due to the selection of the hyperparameters on the same folds as the model assessment. Each fold in both stages is 7 days long. At the end of each iteration, the next fold can be viewed as the current fold shifted forward by one week. The shifting continues until the boundary is reached. All the folds are adjacent to each other in the time series. Each stage of the CV spans 371 days, so that each cross-validation has 53 folds.

A variation of the cross-validation with longer folds is used in the models with long training times. For the experimental setup, a fold can contain multiple weeks of predictions. However, for all the 7-day predictions in a single fold, the model is not retrained. The example of this CV extension is demonstrated in Figure 4.5.

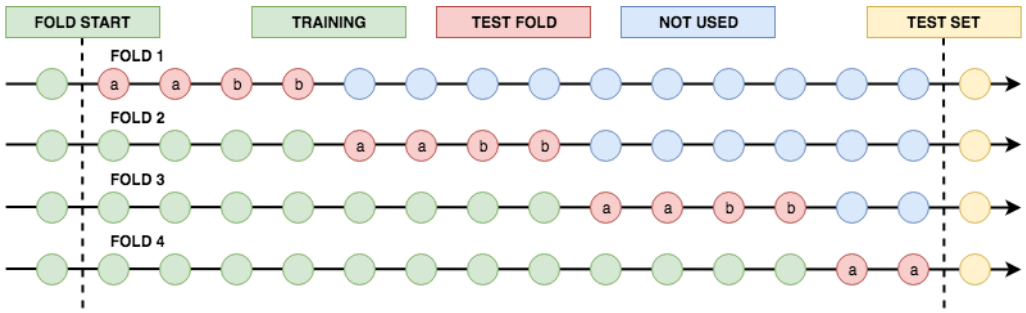

Figure 4.5: Time series 2-step forward rolling origin cross-validation with 4-step long folds.

As evident from the figure, the CV iteration involves 2 predictions of the multistep forecasting model: **a** and **b**. However, the model is not retrained to predict the part **b** of the fold. The part **b** is predicted using the past values, including the observed values in the part **a** of the same fold, but with the model trained on the time series observations before the start of the fold. Next fold is 4 steps forward from the start of the previous fold. When a new fold is reached, the model is retrained using all the observations from

the training part of the time series including parts **a** and **b** of the previous folds. The cross-validation for a 7-day forward forecasting model works the same way, except the fold lengths are a multiple of 7. In the cross-validation with longer folds, all folds have the same length, except the last one, which contains the remaining weeks so that the cross-validation does not step over the validation or test part boundaries.

After the predictions for all folds in the cross-validation have been made, the usual way to summarize the result is to find the average of the errors across all folds. The optimal approach is to find the average of the errors for each non-zero sales day in either the validation or test part of the time series. Each 7-day fold results in up to 7 errors due to zero-sales day skipping, which are then summed up together with the errors from the other folds of the same cross-validation stage, and divided by the number of the days with the positive sales quantity to find the average demand forecast error per day.

4.2.2 Model Performance Summary

4.2.2.1 Error Metrics

When a 7-day forecast is produced in the cross-validation, it is important to know how good this prediction is. In order to do so, some kind of error metric between the observed sales value y_t and the predicted demand $\hat{y}_t$ has to be computed. There is a broad range of possible evaluation metrics applicable to demand forecasts in practice. However, they have their own advantages and disadvantages. One of the most commonly used metrics in time series forecasting is RMSE [5]. Given the sales records $y_i, i \in \{1..N\}$ and model predictions $\hat{y}_i, i \in \{1..N\}$, the RMSE for these N predictions can be summarized as follows:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{4.1}$$

RMSE is known to be scale-dependent [48]. It means that the error depends on the time series variability, and cannot be used for the evaluation of the model performance across different time series, since the final error metric is completely irrelevant as was demonstrated in the M-competition review by Chatfield [14]. Another problem with RMSE is that it is sensitive to outliers. A more robust alternative to RMSE is MAE, or mean absolute error, which is also scale-dependent. Using the previous notation, it can be presented as

follows:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{4.2}$$

Even though both RMSE and MAE have the same units as the time series, MAE is easier to interpret, since MAE is the average absolute difference between the quantity of interest and its forecast unlike RMSE, which involves the sums of squared differences and subsequent normalization with the square root.

Another class of evaluation metrics involves the median of the cross-validation errors. The metrics based on the median are robust against outliers, and provide a good summary of the model performance in 50% of cases. One such metric is MdAE, or median absolute error. Like RMSE and MAE, MdAE is also scale-dependent. It can be specified as follows:

$$\text{MdAE} = \text{median}\big(\,|y_i - \hat{y}_i|\,\big)$$

Due to the scale-dependence of RMSE, MAE, and MdAE, unit-free error metrics are often chosen. One such metric, which is used particularly often, is MAPE [5, 39], which can be presented as follows:

$$\text{MAPE} = \frac{1}{N}\sum_{i=1}^{N}\left|100\frac{(y_i - \hat{y}_i)}{y_i}\right| \tag{4.3}$$

This error metric measures how far the predictions $\hat{y}_i$ are from the observed values y_i relative to y_i in percent. This error metric has multiple problems. One of them is that it is not symmetric. In general, MAPE puts a heavier penalty on the forecasts that are larger than the actual observations [39]. Additionally, this performance metric is infinite if the actual observations are 0. Another problem of MAPE is that it is also sensitive to outliers. In highly volatile time series, MAPE error metric can be easily distorted by the occasional unpredictable spikes in observations. Even though MAPE and MAE are more robust than RMSE in this respect, it would be good if MAPE had the robustness of MdAE. The median absolute percentage error, or MdAPE, can be specified as follows [49]:

$$\text{MdAPE} = \text{median}\Big(\left|100\frac{(y_i - \hat{y}_i)}{y_i}\right|\Big) \tag{4.4}$$

MdAPE is a robust percentage error metric. However, it has the same issues as MAPE with respect to symmetry.

In the experiments, the scale-dependent MAE and RMSE evaluation metrics are used to compare the forecasting models for each time series separately. The MAE is chosen because it is very easy to interpret, and in the case of demand forecasting it represents the exact quantity that we would like to minimize (average error between the demand forecast and actual sales). RMSE is used in addition to MAE in order to monitor the model response to the time series with significant outliers. Most sales histories in the final dataset have unpredictable fluctuations, and it is important to know how well the models handle them.

In addition to the scale-dependent metrics, we use MAPE to track the absolute deviations of the forecasts from the target demand in percent. MAPE allows to summarize the results of a model across all product-location sales time series, and compare these summaries among all models. MAPE is also easy to interpret unlike RMSE or median metrics. In this respect, it is a direct scale-free alternative to MAE.

Sometimes a few large outliers can distort MAE, RMSE, or MAPE error summaries, even though the forecast is excellent for the majority of the days. The insight into the most frequent model behaviour can be gained by checking how large the error is in 50% of cases for the studied models. MdAPE is chosen as an appropriate evaluation metric for this task. The drawback of MdAPE is that it is a little less intuitive than MAE, or MAPE for the model performance interpretation. Like MAPE, MdAPE is scale-free, and can be used to examine the performance of a model across the time series with different variability.

4.2.2.2 Ranking

As mentioned in the previous section, scale-dependent error metrics like MAE and RMSE cannot be used to assess the model performance across many different product-location time series. One way to avoid this limitation is the use of ranking. We can assign the ranks r_i^j to each model j based on how well they perform relative to each other for a product-location time series i according to some error metric. Suppose that the two models F and D have the error metric values a and b for some product-location i. If $a < b$, then $r_i^F < r_i^D$. Therefore, we can summarize the performance of each model in terms of its ranks across all product-locations. In our experimental setting, each model can achieve the ranks from 1 to 7 (since there are 7 models to compare). When the original errors are converted to ranks, the information about the error discrepancy between the ranks is lost. The error difference between the ranks 1 and 2 in product-location i can be multiple times larger than the difference between the ranks 1 and 2 in the product-location j. In

rank comparison, we assume that the model is better if it has a lower rank than another model, while the size of the difference between ranks does not matter.

Given the ranks of the models A and B for 100 product-locations, we can use statistical hypothesis testing to determine if one model is more likely to get better ranks in the specified error metric. In order to do it, we can use the paired sign test due to its simplicity, and minimal assumptions. For the test, it is necessary to have 2 samples, where the observations in both samples are paired. In the case of demand forecasting, the two samples represent the ranks of the compared models, and both samples are for the same error metric. The ranks of the two models are obviously paired by product-location. Given the two models F and D, whose ranks r_i^F and r_i^D belong to the same pair, $i \in \{1, .., 100\}$, the statistical assumptions of the paired sign test are as follows [22]:

1. For all pairs of rank observations (r_i^F, r_i^D), there is the same probability p, $0 < p < 1$, of $r_i^F < r_i^D$.
2. The probability of the event $r_i^F = r_i^D$ is zero.
3. The event $r_i^F < r_i^D$ is independent from the event $r_j^F < r_j^D$, $i \neq j$.

The target of the test is to make conclusions about the parameter p. The second assumption holds because when two regression models of different classes are compared in CV, the chance of their errors being equal is very small and can be disregarded. We use the right tail version of the paired sign test as described in Section 2.3.1. We assume the following hypotheses:

$$\begin{aligned} \mathrm{H}_0 : &\quad p = 0.5 \\ \mathrm{H}_1 : &\quad p > 0.5 \end{aligned}$$

The test statistic T of the sign test is the number of pairs, where $r_i^F < r_i^D$, or $t_{obs} = \#(r_i^F < r_i^D)$. The test statistic follows the Binomial distribution:

$$T \sim \mathrm{Bin}(0.5, 100)$$

The probability $P(T \geq t_{obs})$ gives the p-value. The significance level we use in the experiment is 5%. If the null hypothesis is rejected, it means that $p > 0.5$. This shows that the model F gets a lower (better) rank than the model D with a higher probability for any given fast-moving product time series, which we regard as a superior model. The test can be used for all error metrics, but is especially useful for scale-dependent metrics.

4.2.3 Feature Transformations

Some features in the data have to be preprocessed before the training and forecasting. The general preprocessing step for all models is to scale the weather regressors. It is mostly important for the gradient-based optimization methods used in neural networks and GLM. In order to scale the weather data, the temperature and wind speed features for each PL are divided by their largest respective recordings. In most experimental cases this allows to improve the speed of optimization routine convergence.

The further data transformations depend on the specific model in use. ARIMA-based models work with real numbers, both positive and negative, and require the transformation from the strictly non-negative sales history to the co-domain of real numbers. This is achieved by using the log transformation on the shifted time series. Assume that there exists a non-negative time series $\{y_t\}_{t\in T}, y_t \geq 0$, then the **log transformation** can be presented as follows

$$y'_t = \log(y_t + 1) \tag{4.5}$$

which is a Box-Cox transformation [41] with $\lambda = 0$ of the shifted time series $\{y_t\}_{t\in T} + 1$. The log transformation is also necessary for the neural network models due to a specific problem with the dying rectified linear units (ReLU) discussed further. The log transform is applied to the training part of the time series during cross-validation, and the fold forecasts are then transformed back to the original scale by the inverse of the function in Equation 4.5.

It is important to note that the majority of time series in the dataset are count-based. Forecasting such time series on the continuous scale after transforming them with Equation 4.5 can in theory produce inferior results in comparison to the forecasts of the models like Poisson or negative binomial GLM, which are traditionally used for the time series of counts. However, if the number of counts is large, there is no difference between the continuous and discrete sample spaces [48]. All the time series in this study have large counts as evident from Figure 4.2, where almost all sales histories have at least 1 day with 25 sales. There are no time series with small counts (e.g. from 0 to 5 at the maximum) which are considered to be the primary indication to use the models of counts. Therefore, we assume that the logarithmic transformation should not decrease the quality of forecasts, and the models like dynamic harmonic regression and neural networks will perform at least as well as count-based models.

Another basic feature transformation is needed for the Poisson regression (Poisson GLM). This model requires that the sales response variable is countable during training. However, some sales histories are represented by

the fractional numbers with up to 2 decimal places of precision. In order to solve this problem, the sales variable is multiplied by 100 which is called here the **count transform**:

$$y'_t = y_t \times 100 \tag{4.6}$$

During the cross-validation, Equation 4.6 is applied to the training part of the time series with fractional numbers. After the model has been trained, and the predictions have been obtained, the forecast is divided by 100 and the errors are computed on the original number scale. This approach allows to fulfil the assumption of the Poisson regression about the count response variable. However, the count transform most probably violates another assumption about the equality of the mean and variance of the Poisson GLM [28]. We discuss why this approach is permissible for the subset of the sales time series represented with the real numbers in Section 4.3.3.

4.2.4 Fourier Series Representation of Multiple Seasonality

Many time series in the dataset have more than one seasonality. There is a lack of models that can capture multiple seasonalities out of the box. For example, SARIMA can capture only one seasonality, while ANNs are inefficient when it comes to long seasonalities. There are two general ways how this problem can be solved. The first approach is the time series decomposition described in Section 2.4. The second approach involves seasonality modelling with trigonometric components like sines and cosines [48, 76].

All sales time series in the dataset have at least a weekly seasonality. Other seasonalities are detected with a periodogram, which shows how much frequency components contribute to the time series. The periodogram is computed using the Cooley-Tukey fast Fourier transform algorithm [16]. A typical periodogram is presented as a plot in Figure 4.6.

Using the periodogram for the training set, the pairs (frequency, density) are computed, and then the three pairs with the largest densities are selected. The frequency is then converted into the period for each pair i as a reciprocal. The next step involves checking if the period of any of the three pairs $(\mathbf{period}_i, \text{density}_i), i \in 1, 2, 3$ falls within a certain range:

$$\mathbf{period}_i \in (355, 385) \rightarrow \text{Yearly seasonality present} \tag{4.7}$$

$$\mathbf{period}_i \in (170, 192) \rightarrow \text{Half-yearly seasonality present} \tag{4.8}$$

If it does, then the assumption is made, that a corresponding seasonality in the time series exists. In Figure 4.6, the presented time series has the

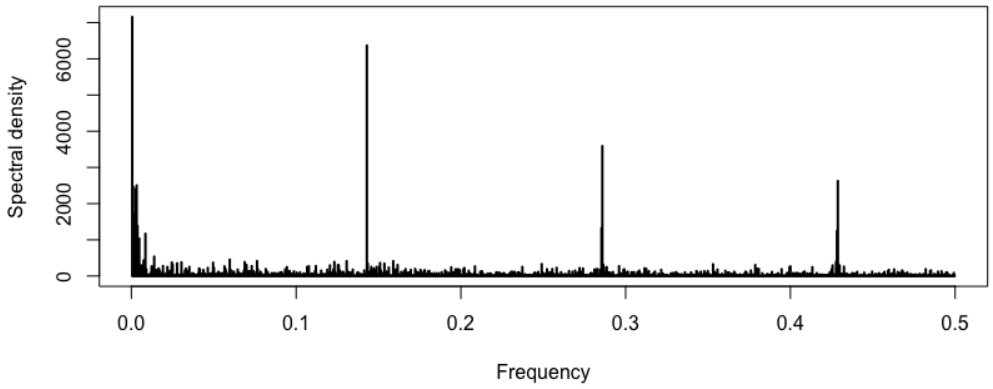

Figure 4.6: Periodogram of a time series with a single weekly seasonality.

three largest periods (corresponding to the tallest spikes in the plot) of 1875, 6.99, and 3.5 days, which results in no additional seasonalities according to the selection criteria (4.7) and (4.8). After the time series seasonalities are discovered, they can be modelled using the Fourier series.

Fourier series is a representation of any periodic function in terms of weighted orthogonal sines and cosines. Suppose that the function $f(x)$ is piecewise continuous and periodic on the interval $x \in [-l, l]$, such that the period $T = 2l$, then the Fourier series representation of the function $f(x)$ can be specified as follows [24]:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \Big(a_n \cos\big(\frac{2\pi n x}{T}\big) + b_n \sin\big(\frac{2\pi n x}{T}\big) \Big) \tag{4.9}$$

$$a_n = \frac{1}{l} \int_{-l}^{l} f(x) \cos\big(\frac{2\pi n x}{T}\big) dx \tag{4.10}$$

$$b_n = \frac{1}{l} \int_{-l}^{l} f(x) \sin\big(\frac{2\pi n x}{T}\big) dx \tag{4.11}$$

where a_n and b_n are Fourier coefficients. In Fourier series analysis, it is important to determine the optimal number of components N, such that the function $f(x)$ is represented sufficiently well. Alternatively, it is possible to smooth the function by choosing a small number of components, since some rapid fluctuations that are described by sines and cosines with higher frequencies, will be effectively left out.

In order to model the weekly seasonality, we introduce 3 Fourier sine terms and 3 Fourier cosine terms as defined in (4.9). For each time step t of the time series $\{y_t\}_{t=1}^{T}$ we compute the Fourier series sines and cosines as follows:

$$\sin\Big(\frac{2i\pi t}{T_w}\Big), \quad i \in \{1,2,3\}$$
$$\cos\Big(\frac{2j\pi t}{T_w}\Big), \quad j \in \{1,2,3\}$$

where $T_w = 7$, which means that the terms represent the weekly seasonality. Each of these sines and cosines computed for all time steps t becomes a separate feature. Likewise, if the periodogram detects either yearly or half-yearly seasonalities, new yearly and half-yearly seasonality features are created from the Fourier terms computed for each time step t. These seasonalities are modelled using only 2 Fourier terms each:

$$\text{Yearly} : \sin\Big(\frac{2\pi t}{T_\text{y}}\Big), \quad \cos\Big(\frac{2\pi t}{T_\text{y}}\Big)$$
$$\text{Half-yearly} : \sin\Big(\frac{2\pi t}{T_\text{hy}}\Big), \quad \cos\Big(\frac{2\pi t}{T_\text{hy}}\Big)$$

where $T_y = 365.25$ and $T_{hy} = 182.6$.

The number of Fourier terms for each periodicity $T_{w/y/hy}$ is kept large enough so that the sines and cosines can represent the underlying seasonality sufficiently well. However, too many components may themselves overfit by adjusting to the noise of the time series too much. For the weekly seasonality of the daily sales time series, the number of Fourier components is kept at six. However, for the yearly and half-yearly seasonalities, it is assumed that the two terms would be enough, since these patterns should be very simplistic and possible to present with at most one sine and cosine. The number of Fourier terms can also be determined with cross-validation or information criteria [48], but for the given dataset, the specified number of seasonal components is assumed to be optimal. Table 4.1 presents the seasonal features of a time series which has both weekly and yearly seasonalities. The weekly covariates are named **w1**, **w2**, **w3**, **w4**, **w5**, **w6**, while yearly features are called **y1** and **y2**.

One of the most interesting properties of the Fourier seasonal modelling is that the Fourier coefficients a_i and b_i are not required to create the seasonality features. These coefficients are in fact learned by the model itself. In the case of linear models, the inference will estimate the regression coefficients corresponding to a set of seasonality features as the Fourier coefficients of the

t	**w1**	**w2**	**w3**	**w4**	**w5**	**w6**	**y1**	**y2**
	$\sin(\frac{2\pi t}{T_w})$	$\cos(\frac{2\pi t}{T_w})$	..	..	$\sin(\frac{6\pi t}{T_w})$	$\cos(\frac{6\pi t}{T_w})$	$\sin(\frac{2\pi t}{T_y})$	$\cos(\frac{2\pi t}{T_y})$
1	0.78	0.62	0.97	-0.22	0.43	-0.90	0.01	0.99
2	0.97	-0.22	-0.43	-0.90	-0.78	0.62	0.03	0.99
3	0.43	-0.90	-0.78	0.62	0.97	-0.22	0.05	0.99
4	-0.43	-0.90	0.78	0.62	-0.97	-0.22	0.06	0.99
5	-0.97	-0.22	0.43	-0.90	0.78	0.62	0.08	0.99
..	..	..	..	..	..	..	..	..

Table 4.1: Weekly and yearly Fourier seasonal features.

smoothed unobserved seasonality component for the given period $T_{w/y/hy}$. In the case of neural networks, not only the coefficients are estimated, but also the number of Fourier terms is adaptively expanded by the neural network model itself because the terms $\cos(nx)$ and $\sin(nx)$ can always be written as non-linear functions of $\cos(x)$ and $\sin(x)$, and this nonlinear dependence can be approximated by the ANN [76].

4.2.5 Modelling Stockouts

As discussed previously, stockout is a situation when the retail store runs out of some product, and the recorded sales drop down to zero if the stock is not immediately replenished. This situation has an effect on demand forecasting, since the sales history on the stockout days does not represent the real demand, which the model ideally should be trained on. For this reason the cross-validation evaluation of the model performance might be wrong. The problem can be solved with a new binary feature called **stockout**.

The **stockout** feature is created for all days of a time series. It is set to 1 for all days when the sales are zero, and 0 otherwise. During training, this feature is used directly by the learning algorithm to distinguish between the stockout and normal days. However, the stockout covariates are set to 0 for the forecast horizon during prediction in cross-validation. This tells the model that the predicted quantities should be the actual demand. Additionally, when the accuracy is finally evaluated, all the forecast days which have the stockout feature set to 1 (days that have 0 recorded sales) are excluded from the accuracy estimation, since the model always forecasts the demand of the days when enough product is available, and comparing its prediction with the real sales data that has a stockout day would be inappropriate. In terms

of cross-validation, it means that in all folds the errors are not computed for the days where the actual sales are 0.

4.3 Regression Models for Demand Forecasting

4.3.1 Naive Baseline Model

Naive forecasting is one of the most commonly used quantitative methods in demand forecasting due to its simplicity. It predicts the demand to be equal to the last available sales history observation. The naive forecasts can include a trend extrapolation, where the trend slope is calculated as the average of the previous two observations. The naive forecasting can also be seasonal, when the future demand is predicted to be the same as the demand on the same day during the previous season. Even though the naive forecasting seems to be too simplistic, it is easy to prepare and interpret, while it has almost no cost [70]. However, the collection of naive methods clearly lacks the means to forecast complicated time series, and account for the presence of noise and complex repeated patterns in data. As a result, the naive forecasting is often used as a benchmark to assess the accuracy and efficiency of other models [48, 70].

Given the simplicity, as well as the speed of the naive forecasting algorithms, the seasonal naive method with a period of one week is chosen as the baseline for the comparison with the other more advanced models. Suppose that the validation/test part of the time series is denoted as $\{y_1^1, .., y_7^1, y_1^2, .., y_7^{53}\}$, where the observation y_i^j is the sales on the weekday i of the fold j, where the folds for the naive model are of length 7, and the model produces 7-day forward multistep forecasts. Assume that $y_1^0, .., y_7^0$ are the 7 sales observations from the end of the training part of the time series if the cross-validation is run for model selection. In the case of final model evaluation, these values would correspond to the last 7 observations of the validation part of the time series. Then, the predictions for the fold j can be given as $\hat{y}_i^j = y_i^{j-1}$. In short, the seasonal naive model forecasts equal the sales values observed the week before. It is expected that all other models in the experiment outperform this naive baseline.

4.3.2 Dynamic Harmonic Regression

Dynamic harmonic regression is a model that consists of two parts: the regression part, and the ARIMA process which represents the errors of the

regression procedure. A data point for this model is presented in Table 4.2.

Features				Label
campaign	**temperature**	**wind speed**	**stockout**	**sales quantity**
w-cos1	**w-sin1**	**w-cos2**	**w-sin2**	
w-cos3	**w-sin3**	**y-cos1***	**y-sin1***	
hy-cos1*	**hy-sin1***			

Table 4.2: Data point for the dynamic harmonic regression.

The data point in Table 4.2 relates the daily sales record with the weather, promotion status and stockout regressors, as well as the Fourier series seasonality features on the same day. The Fourier features are denoted as **w-cos1**, **w-sin1**, **w-cos2**, **w-sin2**, **w-cos3**, **w-sin3**, **y-cos1***, **y-sin1***, **hy-cos1***, and **hy-sin1***. The prefixes **w**, **y**, and **hy** stand for the respective periods of one week, year, and half-year. The numeric ending specifies the multiplier n in the frequency of the sine or cosine as $\cos / \sin\left(\frac{2\pi nt}{T}\right)$. The asterisk means that the presence of the feature is time series specific as described in Section 4.2.4.

Given a sales time series $\{y_t\}_{t\in T}$ with the associated regressor process $\{\boldsymbol{x}_t\}_{t\in T}$ as defined per Table 4.2 for each day t, the dynamic harmonic regression model can be presented in the lag operator notation as follows

$$(1-L)^d y_t = \boldsymbol{\beta}^T (1-L)^d \boldsymbol{x}_t + (1-L)^d \eta_t \tag{4.12}$$

$$\eta_t = \frac{\left(\sum_{j=0}^{q} \theta_q L^q\right)}{\left(1-\sum_{i=1}^{p} \phi_p L^p\right)\left(1-L\right)^d} \epsilon_t \tag{4.13}$$

where $\theta_0 = 1$, η_t is an error term which follows the ARIMA(p,d,q) process, $\boldsymbol{\beta}$ is a vector of regressor parameters of the same dimension as the input feature vector $\boldsymbol{x}_t$, and $\epsilon_t \sim \mathcal{N}(0,\sigma^2)$ represent the innovations used in the maximum likelihood estimation. Equation 4.13 is a MA(∞) representation of the ARIMA process presented in Equation 2.37. By dividing Equation 4.12 with $(1-L)^d$, we arrive at another representation of the dynamic harmonic regression:

$$y_t = \boldsymbol{\beta}^T \mathbf{x}_t + \frac{\left(\sum_{j=0}^{q} \theta_q L^q\right)}{\left(1-\sum_{i=1}^{p} \phi_p L^p\right)\left(1-L\right)^d} \epsilon_t \tag{4.14}$$

By multiplying $(1-L)^d \eta_t$ in Equation 4.12, the term $(1-L)^d$ in (4.13) disappears from the denominator of η_t, which becomes an ARMA(p,q) process representation in terms of MA(∞) as shown in (2.30). From Equation 4.12 it

becomes clear that the differencing can be performed directly on the features $\boldsymbol{x}_t$ and labels y_t to make the error η_t a stationary ARMA process.

The requirement of stationarity still stands for the dynamic harmonic regression errors η_t. However, the situation is harder than in the basic ARIMA model case, since the regression errors are not observable, and cannot be tested for stationarity right away as in a time series without regressors. In order to find the required number of differences d in the autocorrelated part of the dynamic regression model, we first apply the basic OLS regression to the data points ignoring the autocorrelation. Given a time series $(\boldsymbol{x}_t, y_t)$ of length T and regression estimates $\hat{y}_t$, the modified time series $\{\omega_t\}_{t\in T}, \omega_t = y_t - \hat{y}_t$ represents the approximation to the series of autocorrelated errors η_t defined in Equation 4.13. After computing the ω_t values, the time series $\{\omega_t\}_{t\in T}$ can be investigated for stationarity. If this time series requires differencing, then both the labels and the features of the original series $(\boldsymbol{x}_t, y_t)_{t\in T}$ are differenced the required number of times. After the appropriate differencing, the dynamic harmonic regression still includes the regression part, but the ARIMA(p, d, q) errors become the ARMA(p, q) errors. Now, under the assumption of the ARMA innovation ϵ_t normality, the maximum likelihood can be computed, and the parameters of the model can be estimated.

4.3.2.1 ARMA Process State-Space Form

The state-space form, or representation of a dynamic system is a model represented in terms of an observable output $\boldsymbol{y}_t$, known input $\boldsymbol{x}_t$, the hidden state $\boldsymbol{s}_t$ which evolves over time, and the noise components. In a state-space model, the hidden states, the inputs, and the outputs are related by the first-order difference equations. Given a sequence of inputs $\{\boldsymbol{x}_t\}_{t\in T}$, hidden states $\{\boldsymbol{s}_t\}_{t\in T}$, outputs $\{\boldsymbol{y}_t\}_{t\in T}$, as well as sequences of noise terms $\{\boldsymbol{v}_t\}_{t\in T}$ and $\{\boldsymbol{w}_t\}_{t\in T}$, the state space representation of the model dynamics can be specified as follows:

$$\boldsymbol{s}_t = \mathrm{F}\boldsymbol{s}_{t-1} + \mathrm{R}\boldsymbol{v}_t \tag{4.15}$$

$$\boldsymbol{y}_t = \mathrm{A}\boldsymbol{x}_t + \mathrm{B}\boldsymbol{s}_t + \boldsymbol{w}_t \tag{4.16}$$

$$\boldsymbol{v}_t \sim \mathrm{WN}(\mathbf{0}, \mathrm{Q})$$

$$\boldsymbol{w}_t \sim \mathrm{WN}(\mathbf{0}, \mathrm{H})$$

Equation 4.15 is known as the **state/transition** equation, while Equation 4.16 is called the **observation** equation. The vectors $\boldsymbol{v}_t$ and $\boldsymbol{w}_t$ are multivariate white noise, where Q and H are the covariance matrices. Often the state transition equation term $\mathrm{R}\boldsymbol{v}_t$ is written simply as $\boldsymbol{\epsilon}_t$, where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathrm{RQR}^T)$. The matrices F, R, A, B, Q and H specify the model

and contain all the required parameters. In the most basic state-space representation of a model, the following additional assumptions are made [41]:

$$\mathrm{E}\big[\boldsymbol{v}_t\boldsymbol{v}_\tau^T\big] = \begin{cases}\mathrm{Q}, & t=\tau\\ \mathbf{0}, & \text{otherwise}\end{cases} \qquad \mathrm{E}\big[\boldsymbol{w}_t\boldsymbol{w}_\tau^T\big] = \begin{cases}\mathrm{H}, & t=\tau\\ \mathbf{0}, & \text{otherwise}\end{cases}$$
$$\mathrm{E}\big[\boldsymbol{w}_t\boldsymbol{s}_\tau^T\big] = \mathbf{0}, \quad \forall\tau\in T$$
$$\mathrm{E}\big[\boldsymbol{v}_t\boldsymbol{s}_\tau^T\big] = \mathbf{0}, \mathrm{E}\big[\boldsymbol{w}_t\boldsymbol{y}_\tau^T\big] = \mathbf{0}, \mathrm{E}\big[\boldsymbol{v}_t\boldsymbol{y}_\tau^T\big] = \mathbf{0}, \quad \forall\tau<t$$

Any ARMA process can be presented in a state-space form. We use a common representation of the ARMA process by Gardner et al. [33] that can be summarized using Equations 4.15 and 4.16. By also specifying the input matrix A which contains the regression coefficients $\boldsymbol{\beta}$ we arrive at the state-space form that represents the dynamic harmonic regression. Given the model ARMA(p,q), we define $r=\max(p,q+1)$, and set k as the number of features in $\boldsymbol{x}_t$. Then the state-space form matrices for the dynamic harmonic regression can be presented as follows:

$$\mathrm{F} = \begin{bmatrix} \phi_1 & \vdots & & \\ \phi_2 & \vdots & \mathrm{I}_{r-1} & \\ \vdots & \vdots & & \\ \phi_{r-1} & \ddots & \dots \quad \dots & \dots \\ \phi_r & \vdots & \mathrm{O}_{r-1}^T & \end{bmatrix} \quad \mathrm{R} = \begin{bmatrix} 1 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{r-2} \\ \theta_{r-1} \end{bmatrix}$$

$$\mathrm{A} = \begin{bmatrix}\boldsymbol{\beta}_1 & \dots & \boldsymbol{\beta}_k\end{bmatrix}, \ \mathrm{B} = \begin{bmatrix}1 & \vdots & \mathrm{O}_{r-1}^T\end{bmatrix}, \quad \boldsymbol{w}_t = \mathbf{0}, \ \forall t\in T$$

where I_{r-1} is the identity matrix of size $r-1$, and O_{r-1}^T is a row zero vector of length $r-1$. The observation equation noise is set to zero, and does not influence the output. Thus, the observation equation assumes the short form of $\boldsymbol{y}_t = \mathrm{A}\boldsymbol{x}_t + \mathrm{B}\boldsymbol{s}_t$. Depending on the model specification, unless $p=q+1$ in ARMA(p,q), some of the coefficients ϕ_i or θ_j will be zeros. This particular state-space representation is used further for parameter estimation, as well as forecasts in the dynamic harmonic regression model.

4.3.2.2 Kalman Filter and Inference

Given a state space representation of the dynamic harmonic regression model, an algorithm known as Kalman filter can be used to perform parameter estimation and compute forecasts for this model. The Kalman filter is an algorithm that is traditionally used to estimate the hidden state vector $\boldsymbol{s}_t$ in

the state-space models. Given a Gaussian ARMA process, the Kalman filter produces the optimal linear least squares forecasts of the state vector $\boldsymbol{s}_t$ and the next observation $\boldsymbol{y}_t$ based on the past observations of the system up to time t as defined in Equation 2.31. However, its most useful property is that it allows to calculate the exact maximum likelihood function of the ARMA process in a simple way under the assumption that the ARMA innovations are Gaussian [41].

The Kalman filter is an iterative algorithm which consists of the two steps: the update step, and the prediction step. These two steps iterate from the start until the end of the processed sequence. If the past observations and inputs up to step t are given as Ξ_{t-1}, then the state forecast based on these values is denoted as $\hat{\boldsymbol{s}}_{t|t-1}$, and the forecast of the output $\boldsymbol{y}_t$ based on Ξ_{t-1} and $\boldsymbol{x}_t$ is similarly denoted as $\hat{\boldsymbol{y}}_{t|t-1}$. The state forecast $\hat{\boldsymbol{s}}_{t|t-1}$ is associated with the MSE matrix $\mathrm{P}_{t|t-1}$. We define these terms more formally as follows [41]:

$$\hat{\boldsymbol{s}}_{t|t-1} = \hat{\mathrm{E}}\big[\, \boldsymbol{s}_t \,|\, \Xi_{t-1} \,\big] \tag{4.17}$$

$$\hat{\boldsymbol{s}}_{t|t} = \hat{\mathrm{E}}\big[\, \boldsymbol{s}_t \,|\, \Xi_{t} \,\big] \tag{4.18}$$

$$\hat{\boldsymbol{y}}_{t|t-1} = \hat{\mathrm{E}}\big[\, \boldsymbol{y}_t \,|\, \boldsymbol{x}_t, \Xi_{t-1} \,\big] \tag{4.19}$$

$$\mathrm{P}_{t|t-1} = \mathrm{E}\big[(\boldsymbol{s}_t - \hat{\boldsymbol{s}}_{t|t-1})(\boldsymbol{s}_t - \hat{\boldsymbol{s}}_{t|t-1})^T\big] \tag{4.20}$$

where $\hat{\mathrm{E}}$ denotes the optimal linear forecast as shown in Section 2.5.4.1. Given these definitions as well as the state-space form presented in Equations 4.15 and 4.16, the Kalman filter iteration can be specified as follows [41]:

$$\hat{\boldsymbol{s}}_{t|t} = \hat{\boldsymbol{s}}_{t|t-1} + \mathrm{P}_{t|t-1}\mathrm{B}^T(\mathrm{B}\mathrm{P}_{t|t-1}\mathrm{B}^T + \mathrm{H})^{-1}(\boldsymbol{y}_t - \mathrm{A}\boldsymbol{x}_t - \mathrm{B}\hat{\boldsymbol{s}}_{t|t-1}) \tag{4.21}$$

$$\hat{\boldsymbol{s}}_{t+1|t} = \mathrm{F}\hat{\boldsymbol{s}}_{t|t} \tag{4.22}$$

$$\mathrm{P}_{t+1|t} = \mathrm{F}(\mathrm{P}_{t|t-1} - \mathrm{P}_{t|t-1}\mathrm{B}^T(\mathrm{B}\mathrm{P}_{t|t-1}\mathrm{B}^T + \mathrm{H})^{-1}\mathrm{B}\mathrm{P}_{t|t-1})\mathrm{F}^T + \mathrm{R}\mathrm{Q}\mathrm{R}^T \tag{4.23}$$

Given the sequence $\{\boldsymbol{y}_t\}_{t\in T}$ of length T and assuming that the state-space form matrices are known, the Equations 4.21 - 4.23 are iterated until the final state prediction $\hat{\boldsymbol{s}}_{T|T}$ is reached. Then the m-step forward forecast can be computed by the Kalman filter with the following equation [41]:

$$\hat{\boldsymbol{y}}_{T+m|T} = \mathrm{A}\boldsymbol{x}_{T+m} + \mathrm{B}\mathrm{F}^m\hat{\boldsymbol{s}}_{T|T} \tag{4.24}$$

Thus, given the ARMA process specification in the state-space form, the Kalman filter can compute the forecasts for the model. In order to start the iterations, we also need to know the values $\hat{\boldsymbol{s}}_{1|0}$ and $\mathrm{P}_{1|0}$. Their values can be chosen arbitrarily since the Kalman filter will converge to the correct forecasts as it proceeds. However, in order to use the Kalman filter for inference, one

common choice of the hidden state and MSE matrix initialization is the following [33, 41]:

$$\hat{\boldsymbol{s}}_{1|0} = \mathrm{E}\big[\boldsymbol{s}_1\big] = \mathbf{0} \tag{4.25}$$

$$\begin{aligned}\mathrm{vec}(\mathrm{P}_{1|0}) &= \mathrm{E}\big[(\boldsymbol{s}_1 - \mathrm{E}\big[\boldsymbol{s}_1\big])(\boldsymbol{s}_1 - \mathrm{E}\big[\boldsymbol{s}_1\big])^T\big] \\ &= \big(\mathrm{I}_{r^2} - (\mathrm{F} \otimes \mathrm{F})\big)^{-1}\mathrm{vec}(\mathrm{RQR}^T)\end{aligned} \tag{4.26}$$

The operator $\otimes$ denotes the Kronecker product, while the operator vec(.) is the column-wise vectorization operator. This particular initialization has an assumption that the eigenvalues of the matrix F lie within a unit-circle. However, for the stationary ARMA process this is the case by default. Therefore, the initialization in (4.25) and (4.26) is often chosen as the default to start the iterations of the Kalman filter.

The parameters of the ARMA process are usually not known in advance, so only the shapes of the state-space matrices are known (due to the predefined hyperparameters p and q in the ARMA(p, q)). The Kalman filter can be conveniently used to compute the log-likelihood function value of the ARMA(p, q) given that the errors $\boldsymbol{v}_t$ and $\boldsymbol{w}_t$ are Gaussian. In such a case, each observation $\boldsymbol{y}_t$ follows a multivariate normal distribution [41]:

$$\boldsymbol{y}_t|\boldsymbol{x}_t, \Xi_{t-1} \sim \mathcal{N}(\mathrm{A}\boldsymbol{x}_t + \mathrm{B}\hat{\boldsymbol{s}}_{t|t-1}, \mathrm{BP}_{t|t-1}\mathrm{B}^T + \mathrm{H}) \tag{4.27}$$

In the case of sales time series, the output y_t is a single variable, so the distribution in (4.27) reduces to a univariate normal. Therefore, given the sequence of length T of inputs and outputs $\boldsymbol{y} = (\boldsymbol{x}_t, y_t)_{t \in T}$, the log-likelihood function for the Kalman filter can be specified as follows:

$$\begin{aligned}f(\boldsymbol{y}; \mathrm{F},\, \mathrm{R},\, \mathrm{A},\, \mathrm{B},\, \mathrm{Q},\, \mathrm{H}) = \sum_{t=1}^{T} \Big(&- \frac{T}{2}\log 2\pi - \frac{1}{2}\log|\mathrm{BP}_{t|t-1}\mathrm{B}^T + \mathrm{H}| \\ &- \frac{1}{2}(\boldsymbol{y}_t - \mathrm{A}\boldsymbol{x}_t - \mathrm{B}\hat{\boldsymbol{s}}_{t|t-1})^T(\mathrm{BP}_{t|t-1}\mathrm{B}^T + \mathrm{H})^{-1}(\boldsymbol{y}_t - \mathrm{A}\boldsymbol{x}_t - \mathrm{B}\hat{\boldsymbol{s}}_{t|t-1})\Big)\end{aligned} \tag{4.28}$$

The values of $\hat{\boldsymbol{s}}_{1|0}$ and $\mathrm{P}_{1|0}$ are initialized as suggested in (4.25) and (4.26). The log-likelihood is computed by iterating the Kalman filter and computing the parts of the sum in (4.28). Then, all the terms are summed, and the final value of the log-likelihood is returned. This procedure is combined with any optimization algorithm to maximize the function value with respect to the parameter matrices F, R, A, B, Q, H. Most optimization algorithms require the analytic derivatives of the log-likelihood function in Equation 4.28. These can be computed recursively via the Kalman filter iterations [41]. When the

MLE of the matrices is found, it is equivalent to finding the ARMA coefficients of the specified model. The optimization procedure used in the experimental part of this thesis for the dynamic harmonic regression coefficient estimation is Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, which gives the best results in training.

4.3.2.3 Hyperparameter Selection

As noted above, it is important that the error part of the dynamic harmonic regression procedure is stationary. A lot of work has been traditionally focused on the procedures to determine if the time series is stationary or not. Statistical unit root tests such as Dickey-Fuller [21] and Phillips-Perron [63] can be applied to the time series to determine if it is a unit-root process, or trend-stationary. However, these procedures lack statistical power if the tested process is nearly unit root [81]. In order to solve this problem, Kwiatkowski et al. [54] proposes a new statistical test, known as Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test, to determine the presence of a unit root as an alternative to the trend-stationarity. Using the notation from Section 2.3.1:

H_0 : Time series is stationary around the deterministic trend

H_1 : Time series is non-stationary with a unit-root

KPSS is a right-tailed hypothesis test. We use it to determine the necessary number of differences for the dynamic harmonic regression. We first apply the OLS regression to the original time series, and then use the KPSS test with the significance level of 5% to check if the residuals require differencing. If they do, then the residual sequence is differenced, and we try the KPSS test once more. This 2-step procedure is applied repeatedly until the residual series is stationary as indicated by the test (null hypothesis is not rejected), or the upper limit on differences is reached, which is 2 for all the sales time series. Durbin and Koopman [23] note in their textbook that very often real time series cannot be transformed into stationary form with differencing, and it is important to decide when the order of differencing is "good enough", rather than stationary. Thus, we choose the maximum number of differences as 2 which proves to be an excellent choice. Let us denote the number of differences indicated by the KPSS test as d_{kpss}. Since different folds in the cross-validation can have different d_{kpss}, the KPSS test is run for each training part of the 53 folds, and the maximum d_{kpss} is selected as the minimum order of differencing.

The dynamic harmonic regression has 3 hyperparameters: $p, d,$ and q. We treat the number of differences $d = \{d_{\text{kpss}}, d_{\text{kpss}} + 1\}$ as a hyperparameter

with 2 values in order to detect if the KPSS test made a type 2 error, and more differences would result in a better fit. The ARIMA part modelling involves also the selection of the AR and MA orders p and q. Hyndman [48] suggests that Akaike's Information Criterion (AIC) is suitable for this task. However, AIC is inappropriate when ARIMA models with different orders d are compared [48]. Thus, the procedure of choice is cross-validation, which can help select both the AR and MA terms, as well as the differencing term d. The hyperparameter space for the cross-validation is specified on the grid. The set of available hyperparameter combinations of ARIMA(p, d, q), where $P = \{0, .., 7\}, Q = \{0, .., 7\}$, and $D = \{d_{\text{kpss}}, d_{\text{kpss}} + 1\}$, is specified as a Cartesian product $P \times D \times Q$. Thus, 128 combinations of hyperparameters are tried in cross-validation. The four best ARIMA models are selected based on MAE, RMSE, MAPE, and MdAPE for each time series.

4.3.3 Poisson Regression with Elastic Net

The majority of sales histories in the dataset are time series of counts, that can be modelled with Poisson regression. Poisson regression is a GLM which has the Poisson distribution as its random component, and the log-link function. The Poisson distribution can be rewritten in the exponential family form defined in (2.17) as follows

$$f(y; \lambda) = \frac{1}{y!} e^{\frac{\log(\lambda) y}{1} - \lambda} \tag{4.29}$$

where $\log(\lambda)$ is the canonical parameter for the Poisson class of distributions, and the dispersion parameter is simply 1. Since the logarithm appears in the canonical parameter, logarithmic link is a canonical link function. It means that the log-likelihood of the Poisson regression can be efficiently maximized with the Newton's method.

Given a data point $(\boldsymbol{x}_t, y_t)$, where $\boldsymbol{x}_t$ is a feature vector, and y_t is a label, the Poisson regression model for one data point can be specified as follows:

$$f_{Y_t|X_t}(y_t|\boldsymbol{x}_t) = \frac{e^{-e^{\boldsymbol{\beta}^T \boldsymbol{x}_t}} e^{y_t \boldsymbol{\beta}^T \boldsymbol{x}_t}}{y_t!}$$

where $\boldsymbol{\beta}$ denotes the vector of GLM coefficients. The mean of the conditional Poisson distribution is $\mathrm{E}\big[\, Y_t \,|\, X_t = \boldsymbol{x}_t \,\big] = e^{\boldsymbol{\beta}^T \boldsymbol{x}_t}$. The log-likelihood of the Poisson regression for T data points $(\boldsymbol{x}_t, y_t)$, where it is assumed that $y_t|\boldsymbol{x}_t$ are i.i.d., is presented as follows:

$$\mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{T} \left(y_t \boldsymbol{\beta}^T \boldsymbol{x}_t - e^{\boldsymbol{\beta}^T \boldsymbol{x}_t} - \log(y_t!) \right) \tag{4.30}$$

One common concern when applying Poisson regression is the violation of the assumption that the mean of the distribution equals its variance. In the context of Poisson regression, overdispersion means that the variance of the data is larger than what would be expected under the model assumption of the equal mean and variance, and underdispersion is a smaller variability in the data than the model expects. Therefore, Poisson regression is not an optimal model for overdispersed or underdispersed data. In practice, the overdispersion is encountered much more often than the underdispersion, and there exist tests to detect it [17]. However, small amounts of overdispersion do not influence the efficiency of the maximum likelihood estimation, so the model can be applied with minor assumption violations [17].

In the case of overdispersion or underdispersion, a specific kind of estimator called Poisson quasi-likelihood function introduced by Wedderburn is often used [74]. The quasi-likelihood function is generally not a likelihood function, but in the case of a single parameter Poisson distribution, the quasi-likelihood function is equal to the log-likelihood of the Poisson GLM [74]. The quasi-likelihood effectively deals with overdispersion and underdispersion, if the variance is proportional to the mean, which is usually a reasonable assumption. One beautiful fact about the quasi-likelihood estimates of Poisson regression coefficients $\boldsymbol{\beta}$ is that they are identical to the standard maximum likelihood estimates of the Poisson GLM [28, 74]. It is important to note that the estimated coefficient confidence intervals differ [28]. However, in this work we are only interested in the best predictions of the Poisson GLM model, and disregard the confidence bounds on the estimate of the regression coefficients $\boldsymbol{\beta}$. Therefore, the Poisson maximum log-likelihood estimation on over- or underdispersed data gives the same results as the estimation of coefficients with the Poisson quasi-likelihood function. Since the quasi-likelihood should deal with the violation of the mean-variance equality assumption out of the box, the prediction quality of the Poisson GLM is not influenced by the overdispersion or underdispersion in the time series if the variance is proportional to the mean which we readily assume further.

In Equation 4.6 we introduced the count transform for the time series with real numbers. We note, that when the random variable is scaled with some coefficient α, its mean changes linearly as $\mathrm{E}\big[\alpha X\big] = \alpha \mathrm{E}\big[X\big]$, while its variance is scaled by the square of the multiplier, $\mathrm{Var}\big(\alpha X\big) = \alpha^2 \mathrm{Var}\big(X\big)$. Unless the variance was α times larger than the mean before the transformation, the output time series is most probably underdispersed. However, we still assume that the variance is proportional to the mean. Under this assumption, we use the result about the Poisson quasi-likelihood, which says that even if the time series is underdispersed, the estimate of the coefficients $\boldsymbol{\beta}$ with the Poisson GLM log-likelihood will still be appropriate for prediction. Thus,

we can safely use the count transformation to convert the fractional number time series into the count time series. In terms of the original problem, it means we are trying to predict the demand in batches of 10 grams instead of some fraction of a kilogram.

The sales time series have a large number of outliers, and very often the patterns are erratic and likely to be influenced by some unknown factors. At the same time, we would like to reduce the model complexity so that it does not adjust to the noise too much. For this purpose, a specific type of regularizer called elastic net is introduced [83]. The elastic net regularization combines the L_2 and L_1 penalties defined in Equations 2.22 - 2.23, and inherits the properties of both: it drives the regression coefficients to zero as in the ridge regression, and tries to create a sparse representation of the coefficient vector $\boldsymbol{\beta}$ as in the LASSO. In addition, the elastic net has a very useful grouping property such that highly correlated predictors are set to zero or used in the prediction simultaneously [83]. The log-likelihood of Poisson regression model with elastic net regularization can be specified as follows:

$$\mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{T} \Big(y_t \boldsymbol{\beta}^T \boldsymbol{x}_t - e^{\boldsymbol{\beta}^T \boldsymbol{x}_t} - \log(y_t!) \Big) - \lambda \Big(\alpha \|\boldsymbol{\beta}\|^2 + (1-\alpha) \sum_{i=1}^{S} |\boldsymbol{\beta}_i| \Big) \tag{4.31}$$

where the parameter λ controls the strength of the regularization, T is the number of data points, S is the size of vector $\boldsymbol{\beta}$, and the parameter $\alpha \in (0, 1)$ specifies the share of the regularization provided by the L_2 penalty. These 2 parameters are selected manually.

4.3.3.1 Model Training

The demand y_t is dependent on the vector of features $\boldsymbol{x}_t$ which includes the temperature, wind speed, stockout and campaign indicators, as well as the seasonal Fourier terms on date t. In order to model the time dependency of sales observations, the sales history preceding the date t (autoregressive terms) is included in the feature vector. The number of the autoregressive terms (window size) in $\boldsymbol{x}_t$ can range from 1 to 60. The window size p is tuned as a hyperparameter. Table 4.3 demonstrates how a full data point at t can be presented as a vector of features and a related label.

The feature vector $\boldsymbol{c}_t$ is a section of $\boldsymbol{x}_t$ which contains the regressors for the date t, and can include from 6 to 10 Fourier terms, depending on the number of seasonalities in the time series as described in Section 4.2.4. The feature names in Table 4.3 follow the convention introduced earlier in Section 4.3.2.

Autoregressive Features					Current Features	Label
y_{t-p}	y_{t-p+1}	..	y_{t-2}	y_{t-1}	$\boldsymbol{c}_t$	y_t

$\boldsymbol{c}_t$			
campaign	**temperature**	**wind speed**	**stockout**
w-cos1	**w-sin1**	**w-cos2**	**w-sin2**
w-cos3	**w-sin3**	**y-cos1***	**y-sin1***
hy-cos1*	**hy-sin1***		

Table 4.3: Full data point for Poisson regression.

Suppose that the forecast has to be produced for 7 days forward starting at date s. The real sales values are from y_s to y_{s+6}. The training portion of the time series ends right before the time step s. If the model window size is p and the time series starts at $t = 1$, then for each $t \in \left[\, p+1, s-1 \,\right]$ we create a feature vector $\boldsymbol{x}_t$ and its label y_t. In order to create a training data point at index t, we take the previous sales values from $t-p$ to $t-1$ inclusive (autoregressive features), and concatenate them with the features $\boldsymbol{c}_t$ at time t as shown in Table 4.3. The label for this feature vector is y_t. Thus, the training set for the model which predicts the fold at time step s will consist of $s-p-1$ data points. The procedure is illustrated in Figure 4.7. In the picture, the constituent parts of the training data point at the time step $s-1$ are illustrated with the dashed border: p autoregressive terms are gathered in the left rectangle, while the features $\boldsymbol{c}_{s-1}$ are in the right vertical rectangle with the label y_{s-1}. The time series section used to prepare the training set is colored green, while the test fold is colored red.

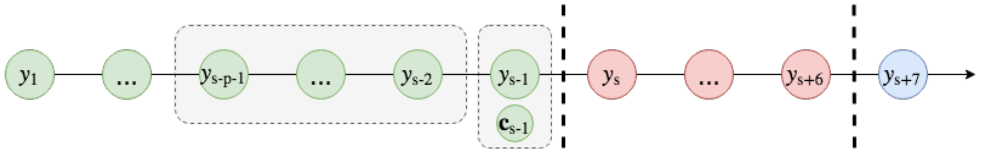

Figure 4.7: Training set preparation for Poisson GLM.

After the training set has been created, the model parameter estimation can take place. The regularization hyperparameters λ and α must be selected in advance. The model is trained with respect to the vector $\boldsymbol{\beta}$ using a two-step optimization procedure implemented in the R package **glmnet** [43] which is a non-standard implementation of the Newton's method using a quadratic log-likelihood approximation and coordinate descent optimization

algorithm [30]. In order to use this procedure, the maximum likelihood function in Equation 4.31 is redefined as a loss function:

$$\mathrm{L}(\boldsymbol{\beta}; \mathbf{X}) = -\frac{1}{T}\mathcal{L}_Q(\boldsymbol{\beta}) + \lambda\Big(\frac{(1-\alpha)}{2}\|\boldsymbol{\beta}\|^2 + \alpha\sum_{i=1}^{S}|\boldsymbol{\beta}_i|\Big) \tag{4.32}$$

where $\mathbf{X}$ is the training set features and labels, and $\mathcal{L}_Q(\boldsymbol{\beta})$ stands for the quadratic expansion around the Poisson log-likelihood function in Equation 4.30. The log-likelihood approximation $\mathcal{L}_Q$ is divided by the number of the training examples T. The parameter α corresponds to the share of the LASSO regularization in the elastic net. The complete optimization routine for a given λ can be specified as a combination of two steps as follows [30]:

1. Perform a quadratic expansion of the Poisson log-likelihood function in Equation 4.30 as described in Equation 2.18

2. Append the elastic net penalty to the quadratic log-likelihood approximation to form Equation 4.32 and run the coordinate descent to optimize for the $\boldsymbol{\beta}$ until convergence

These 2 steps together represent one iteration of the Newton's method as described in Equation 2.19. The step-size is not controlled in the implementation [30], and it might influence the estimation negatively. However, in the experimental part it was noticed that the algorithm always converges well. The coordinate descent procedure is conceptually similar to the gradient descent, except that at each iteration only 1 parameter is updated with its partial derivative (full gradient is not computed), while the other parameters have to wait their turn. This specification severely limits the coordinate descent since it cannot be effectively parallelized (one coordinate update has to be done sequentially with other coordinate updates), and sometimes it can circle around the optimum without convergence. However, this optimization method is preferred for the Poisson GLM with elastic net since the residual vectors in the loss function minimization become sparse, and far fewer parameters have to be updated, which makes the coordinate descent much faster in estimation than all other alternatives [30].

After the model has been trained, the fold test values should be compared to the model predictions. Suppose that the dates $s + a$, $a \in \{0, .., 6\}$, correspond to the test fold. The test day value y_s is easy to forecast given the training portion of the time series since all immediate autoregressive terms are known. However, for $a > 0$, we need the values $y_s, y_{s+1}, .., y_{s+a-1}$, that are not known in advance. One way to solve this problem is to forecast recursively from s to $s + 6$, while substituting the forecasts $\hat{y}_{s+a}$ for the real

values y_{s+a}, where $0 \leq a < 7$. Thus, the autoregressive part of the feature vector $\boldsymbol{x}_{s+a}$ will include all known time series values up to time step s, while the forecasts $\hat{y}_{s+a}$ are substituted for the real values y_{s+a} which are not known beforehand. Using this technique, the Poisson GLM with elastic net produces the multistep forecasts.

4.3.3.2 Hyperparameter Selection

Poisson GLM with elastic net has three important hyperparameters:

1. Regularization strength parameter $\boldsymbol{\lambda}$
2. LASSO share $\boldsymbol{\alpha}$ in the total elastic net penalty
3. Window size $\boldsymbol{p}$, or number of autoregressive terms in the feature vectors

The cross-validation with the grid search is used to choose the best hyperparameters. The windows size p is an integer from 1 to 60, defined as a set $W = \{1, 2, .., 59, 60\}$. The parameter α is selected from 6 possible options, $A = \{0, 0.2, 0.4, 0.6, 0.8, 1\}$. In the case $\alpha = 1$, only the L_1 penalty is used, and if $\alpha = 0$, only the L_2 penalty is used. Thus, the ridge and LASSO regularization procedures are just two specific cases of the more general elastic net penalty.

Hyperparameter	Range
λ	$\{\lambda \mid \lambda = e^s, s = i * l + \log(0.00001), 0 \leq i \leq 20\}$ where $i \in \mathbb{Z}$ and $l = \frac{\log(10) - \log(0.00001)}{20}$
α	$\{0, 0.2, 0.4, 0.6, 0.8, 1\}$
p	$\{1, 2, 3, .., 58, 59, 60\}$

Table 4.4: Elastic net Poisson GLM hyperparameter ranges for the grid search.

The regularization strength parameter λ can be anywhere from 0.00001 to 10. However, the values in the lower range are as important to check as the values larger than 0.1. In fact, we cannot simply divide the parameter range into n equidistant points, since it will only check the lambdas in the range above 0.4, while the optimal regularization strength parameter values are often below 0.1. In order to solve this problem, the logarithm is applied to the upper and lower limits of the range of λ. As a result, the range of λ on the logarithmic scale extends from 2.30259 to -11.51293. We pick 20

equidistant points $\log(\lambda)$ from this interval such that the whole logarithmic range is covered. Then the sample points are converted back to the normal range with exponentiation. We denote this set of suggested λ values as Λ. The hyperparameter ranges are summarized in Table 4.4. The grid search produces $W \times A \times \Lambda$ hyperparameter combinations to try. The parameter estimation in the Poisson GLM with elastic net using **glmnet** is very fast, and the grid search procedure for hyperparameter selection is not prohibitive.

4.3.4 Deep Neural Networks

In the past, the applied research in time series forecasting traditionally used neural networks with shallow architectures. With the advent of deep learning, new NN models with multiple layers and recurrent connections are now applied to time series forecasting too. In this work we investigate the effectiveness of deep MLP (two hidden layers) and LSTM (two stacked layers) architectures applied to demand forecasting in grocery retail. The seasonal features are modelled with Fourier terms. However, we also separately test the MLP and LSTM models with completely deseasonalized time series. In the work of Chu and Zhang [15] it was found that the deseasonalization works better than trigonometric modelling in neural networks. We would like to verify this claim. Thus, the number of the evaluated neural network models is 4: MLP and LSTM-based models with Fourier seasonal modelling, as well as MLP and LSTM-based models with deseasonalized time series.

4.3.4.1 Seasonal Decomposition

As suggested in Section 4.2.4, we preprocess the time series data to create new seasonality features represented by a set of trigonometric components. This is the basic mode of MLP and LSTM evaluation. We also create 2 separate models: one additional MLP and LSTM model. They work with the time series without Fourier features. The seasonality is subtracted from the log-transformed time series before processing. After the model makes the predictions, the seasonal component is added on top of them to produce the final forecast.

The seasonal decomposition (described in Equation 2.14) is done with the moving averages. If the test fold in cross-validation starts at step s of the time series $\{y_t\}_{t>0}$, then the seasonal component is estimated from the time series part before s. We compute the moving averages with the window size of 7 days. For each point in the time series from $t = 4$ to $t = s - 4$, the averages $\omega_t = \frac{1}{7}\sum_{i=-3}^{3} y_{t+i}$ are computed. The offsets are needed since there are no values beyond $t = 0$ or $t = s - 1$, and the full moving average window

centered on the border cannot be estimated. After the averages ω_t have been computed, they are subtracted from the time series to form a new sequence $y'_t = y_t - \omega_t$. If $0 < t < 4$ and $s > t > s - 4$, then $y'_t = y_t$. To form a seasonal component C, we compute the averages of the new series y'_t for each day of the week $1 \leq j \leq 7$ with a skip of length 7:

$$C_j = \frac{1}{\lfloor \frac{s-1-j}{7} \rfloor} \sum_{i=0}^{\lfloor (s-1-j)/7 \rfloor} y'_{j+7*i} \tag{4.33}$$

The floor operator is applied to the result of the division to find the number of full weeks after the first occurrence of day j. The seasonal component C is a vector of length 7. The final deseasonalized time series is produced by subtracting the estimated seasonal component C from the time series $\{y_t\}_{0<t<s}$. The deseasonalized time series $\{y_t^{\text{noseason}}\}_{0<t<s}$ can also be converted back to the original time series by applying the reverse operation. The equations are as follows:

$$y_t^{\text{noseason}} = y_t - C_{(t-1)\%7+1} \tag{4.34}$$

$$y_t = y_t^{\text{noseason}} + C_{(t-1)\%7+1} \tag{4.35}$$

where % denotes the modulo operation. The MLP and LSTM models that operate on the deseasonalized time series $\{y_t^{\text{noseason}}\}_{t \in T}$ do not need the Fourier features anymore. If the forecasts for the test fold from s to $s + 6$ are needed, then the NN model is trained on the deseasonalized time series $\{y_t^{\text{noseason}}\}_{0<t<s}$, and the fold predictions $\hat{y}_k^{\text{noseason}}, s \leq k \leq s+6$ are produced. Then Equation 4.35 is applied to the predictions $\hat{y}_k^{\text{noseason}}$ which gives the final demand forecast, which is compared against normal sales values y_k in the cross-validation.

4.3.4.2 Multilayer Perceptron

The basic neural network model for demand forecasting is multilayer perceptron discussed in Section 2.6. MLP is a very popular tool for time series forecasting implemented in many packages and often mentioned in research. In order to forecast future sales, we use an MLP with 2 hidden layers as shown in Figure 4.8. This is a deep learning architecture since it uses 2 hidden layers in comparison to a shallow architecture with 1 hidden layer.

In Figure 4.8, the number of neurons in the first hidden layer is denoted as a, and the number of neurons in the second hidden layer as b. The activation function in both hidden layers is the leaky rectifier as described in Equation 2.43. The output layer activation function is linear, and requires

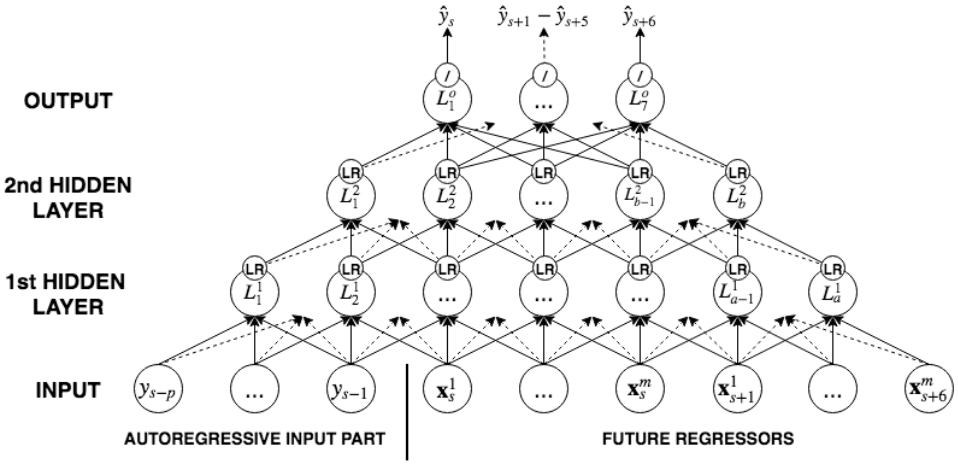

Figure 4.8: MLP architecture for the 7-day demand forecasting task.

the log-transformation of the sales feature as described in Equation 4.5. The rectifier activation function (2.42) for the output is a more natural choice, since the demand is always non-negative. However, the rectifier is not used by default since it causes the dying ReLU problem in some time series, which is discussed further.

We apply the batch normalization after each hidden layer. The leaky rectifier and the batch normalization applied after it are denoted with the **LR** abbreviation for each respective neuron in Figure 4.8. The batch normalization is used in our 2-layer MLP architecture because it improves the training times for the sales time series dramatically. At the same time, ReLUs are a popular choice for deep neural networks (especially in image recognition and computer vision) since they allow NNs to achieve the best performance without unsupervised pretraining [37]. This likely happens due to the ability of ReLUs to create sparse representations of input data. However, rectifiers also reduce the model capacity [37].

One common problem of the rectifier activation function is dying ReLUs [57]. The neurons with the rectifier activation are considered dead if at some point during training their linear response $\mathbf{a}$ becomes negative for every possible training batch. The negative linear response means that the output of the rectifier activation is always zero, and its derivative is zero too. In turn, zero derivative means that the backpropagation routine will not update the neuronal weights past the ReLU with the negative linear response anymore. Dying ReLU is a type of the vanishing gradient problem. When a ReLU dies, it automatically reduces the model capacity: a few dying ReLUs

can have negligible influence, while many will result in unsatisfactory performance. The dying ReLU issue is usually encountered when the learning rate is too large. In terms of the sales time series examined in this work, the time series with artifacts depicted in Figure 4.9 cause the dying ReLU situation. The characteristic property of such a time series is that the number of sales is usually moderate, but sometimes the outliers occur. These outliers result in large forecast errors during training, that set the linear response of the ReLUs negative in backpropagation. The problem can be avoided by using a much smaller learning rate. However, the training is not efficient in that case since too many epochs are needed to reach convergence.

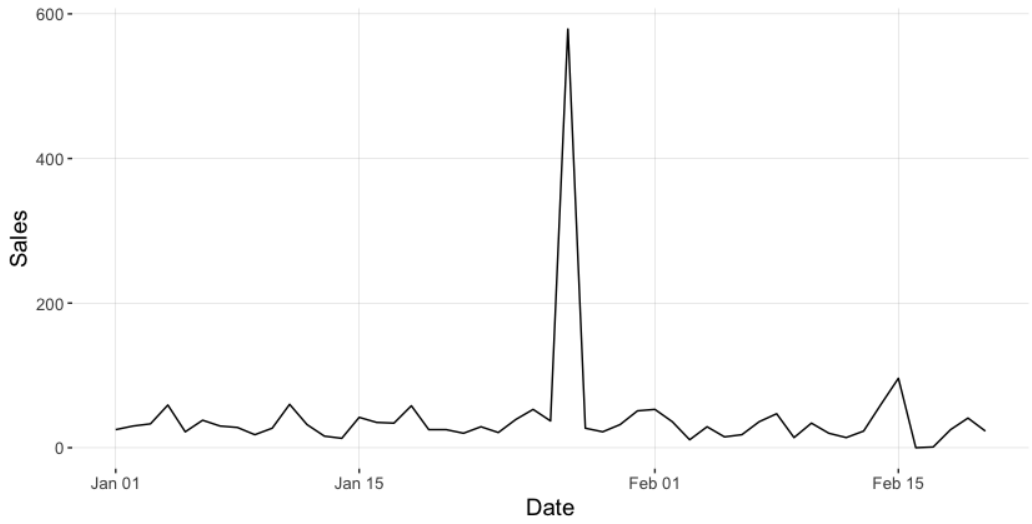

Figure 4.9: An example of time series causing the dying ReLU.

In our architecture the dying ReLU problem is solved by using leaky ReLUs instead of normal ReLUs. Leaky rectifier activation functions allow to preserve the better convergence property of the rectifier activation function [79], but do no cause the dying ReLU phenomenon since the gradient is never zero for the negative neuronal linear response.

Our MLP architecture produces the multistep forecast $\hat{y}_s$ - $\hat{y}_{s+6}$ in one forward pass through the network. This is achieved by designating 7 neurons as the output layer of the MLP as shown in Figure 4.8. The input vector of the MLP consists of 2 parts: the autoregressive part, and the future regressor part. Since the prediction is obtained for the whole forecast horizon at once, the input vector should contain the regressors for all 7 future days. The length of the autoregressive part (window) is determined by the cross-validation procedure. Suppose that the forecast is made for 7 days forward starting from

date s. Denote the external regressors for date t as $\boldsymbol{x}_t$. If we use the Fourier seasonal modelling, the vector $\boldsymbol{x}_t$ consists of the following features: campaign and stockout binary indicators, weather variables, and from 6 to 10 features corresponding to the available seasonalities as described in Section 4.2.4. If we use the deseasonalization as described in Section 4.3.4.1, then the number of features in the regressor vector $\boldsymbol{x}_t$ is 4. The MLP is trained and evaluated for both types of seasonality modelling as two separate models. However, there is no architectural difference between the two MLPs except for the input layer size.

Given the test fold at step s and the window size p, the sales values from y_{s-p} to y_{s-1} are taken as the autoregressive part of the input vector. Next, we take the seven vectors of external regressors corresponding to the future demand forecast, $\boldsymbol{x}_s$ to $\boldsymbol{x}_{s+6}$. They are packed into 1 vector by concatenation. The resultant vector $\boldsymbol{x}_{s:(s+6)}$ is joined with the vector of the autoregressors y_{s-p} - y_{s-1}. The parts of the MLP input vector are illustrated in Figure 4.10. In the picture, the red values indicate that the respective components are not

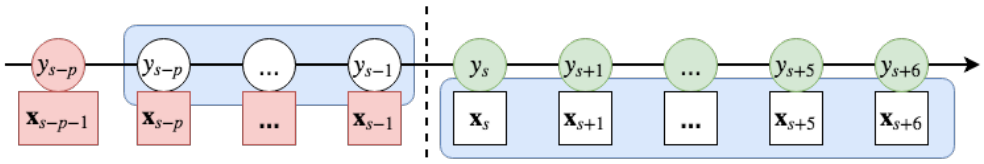

Figure 4.10: Input vector values parsed out from the time series before the fold forecast.

used, blue background emphasizes that the component gets concatenated into the input vector, and the green colour denotes the fold values for which the forecasts have to be produced.

The training set of the MLP model is composed of the input vectors which are produced as described above, as well as the labels which are the vectors of length seven. Given that the window size is p and the test fold starts at s, the training set will contain $s - p - 7$ data points. The parameter p is subtracted since the first training input has to start at $t = 1$, which means that the effective start of the first label will be at $p+1$. The forecast horizon is 7, so the last label vector will start at index $s-7$. The training data points are produced for all t, $p + 1 \leq t \leq s - 7$. The procedure can be presented as a window of size $p + 7$, which is moved along the training part of the time series, cutting out the input vectors at each stop, until the test fold is reached.

The MLP model can easily overfit given sufficiently long training times,

and the regularization is required. We use the dropout in the second hidden layer, and the early stopping. We choose the dropout over the L_1 or L_2 regularization, since all three methods produce comparable results, but it is easier to find the right regularization strength for the dropout since its hyperparameter ranges from 0 to 1. The use of the cross-validation complicates the use of early stopping, since it can indicate a different number of epochs for each fold. In order to solve this problem, we predetermine the maximum number of epochs in training for all folds during the model selection cross-validation. Every 50 epochs, we measure the test performance of the MLP in each fold, and record this value. At the end of the cross-validation, we have the records of errors across all folds that correspond to a specific number of epochs. We average the fold errors across all folds for each number of epochs separately, and select the number of epochs which corresponds to the lowest average error. This number of epochs is used further on the test set to evaluate the model. Thus, the number of epochs is treated as a hyperparameter.

The tuned hyperparameters for the multilayer perceptron are the size of the sliding window (number of autoregressive terms p), the number of neurons in the first hidden layer, the number of neurons in the second hidden layer, the slope of the leaky rectifier in the two hidden layers, the dropout regularization strength in the second hidden layer as well as the number of epochs in training. The ranges of these hyperparameters are summarized in Table 4.5.

MLP Hyperparameter	Range/Set
Window size (**ws**)	$\{3, 7, 14, 28\}$
Neurons, first hidden layer	from $(\mathbf{ws} + 84)$ to $2 \times (\mathbf{ws} + 84)$
Neurons, second hidden layer	from $\lfloor(\mathbf{ws} + 84)/2\rfloor$ to $(\mathbf{ws} + 84)$
Slope of the leaky rectifier	$\{0.1, 0.3, 0.5\}$
Dropout	$\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$
Number of epochs in training	$\{200, 250, 300, 350, 400, 450, 500, 600\}$

Table 4.5: Two layer MLP hyperparameter ranges used in cross-validation.

4.3.4.3 Long Short-Term Memory

Another model particularly suitable for sequence processing is LSTM. The traditional LSTM cell as presented in Section 2.6.2 can be thought of as a neural network with one layer. Even though an RNN like LSTM can be

unwrapped through time and presented as an infinitely deep neural network depending on the length of the processed sequence, the purpose of its layers is to introduce memory, while in conventional deep learning the layers serve the purpose of processing the inputs hierarchically [44]. In fact, the traditional LSTM learns through time what bits of input to keep, and which to discard. The deep learning on the contrary is concerned with hierarchical input representations learned by the neural network in between layers, which is not done in one-layer LSTM. Thus, we use a two-layer LSTM design which satisfies the criteria of a deep learning LSTM architecture [61]. The architecture is presented in Figure 4.11, where the horizontal direction is aligned with the time.

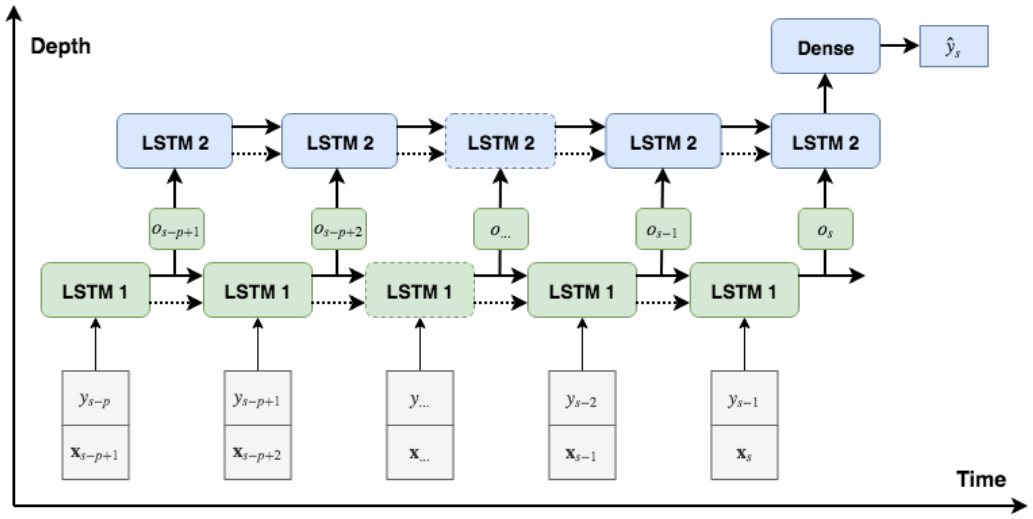

Figure 4.11: Two-layer LSTM architecture for sales forecasting.

In Figure 4.11, the first LSTM layer is denoted with the code **LSTM 1** (green color), and the second layer with the code **LSTM 2** (blue color). Each layer is a separate LSTM which has been unwrapped through time as described in Section 2.6.2. The LSTM in each layer has two recurrent connections: the memory cell, or internal state, and the output. The LSTM internal state is denoted with the dotted line, while its output is drawn with a solid black line. From the figure it is clear that the outputs o_t (colored green too) of the first layer are fed into the second LSTM layer. The purpose of the LSTM in the first layer is to transform the original input sequence into a higher level sequence of representations which are easier to digest for the second layer LSTM.

On top of **LSTM 2**, there is a **Dense** layer. The **Dense** layer is a

single neuron with the linear activation function to transform the output of the **LSTM 2** at the last time step into the right form. As shown in Figure 2.9, the output of the LSTM has the same dimensionality as the LSTM cell, and each vector entry has the range from -1 to 1 due to the tanh-transformation applied to the cell state right before the output. The **Dense** layer can reduce the dimensionality of the output as well as change the range of the **LSTM 2** output values. The **Dense** layer does not serve any other meaningful purpose, and can be thought of as a part of the LSTM output channel. The time series sales variable has to be log-transformed before this model can be applied to it.

The LSTM in the second layer does not produce an output at each time step. In the case of our deep LSTM architecture, we forecast sales only 1 day forward for a given sequence. If the forecast is required for the day s, the output of the neural network in Figure 4.11 is denoted as $\hat{y}_s$. Only the last output of the second layer is accepted as a forecast. The second LSTM layer has its training exclusively through the memory cell, and the equations in the gradient descent procedure do not involve intermediate outputs $\hat{y}_{t|t<s}$. The first layer LSTM unit is trained with all input-output pairs in the sequence, where at each time step the error is backpropagated from both the future and the upper layer LSTM.

The input of the two-layer LSTM network at time t is the sales value of the previous time step y_{t-1} and the vector of regressors $\boldsymbol{x}_t$, which consists of the two weather variables (wind speed, temperature), as well as campaign and stockout indicators: 5 basic features altogether. If the seasonality is modelled using the Fourier series, then 6 to 10 seasonal sine/cosine terms are added to $\boldsymbol{x}_t$ as described in Section 4.2.4. We define the input vector for the LSTM at time step t as $\boldsymbol{\xi}_t = (y_{t-1}, \boldsymbol{x}_t)$, a concatenation of the sales value y_{t-1} and the regressors $\boldsymbol{x}_t$. In order to predict y_s, we need to provide a sequence of inputs for the LSTM-based NN, so that the cells in both LSTM layers can remember the values $\boldsymbol{\xi}_{t|t<s}$, and produce the forecast $\hat{y}_s$. Therefore, the forecast value y_s is the label, while its features are a two dimensional array, or matrix, where one of the dimensions is the features $\boldsymbol{\xi}_t$, while the other dimension is the time steps t. The data point for a 2-layered LSTM is demonstrated in Table 4.6. By using the previous day sales record y_{t-1} in each input $\boldsymbol{\xi}_t$, the presented LSTM architecture can process external regressors without the need to bridge it with an MLP construct, or duplicate features.

Even though the training set can theoretically contain a single input matrix which is equal to the whole multivariate time series up to time s, LSTM-based architectures have severe limitations in terms of the sequence length due to the speed of backpropagation optimization. A more traditional approach is to split the time series $\{y_{t-1}, \boldsymbol{x}_t\}_{t<s}$ into subsequences with a sliding

Feature Matrix					Label
y_{s-p}	y_{s-p+1}	..	y_{s-2}	y_{s-1}	y_s
$\boldsymbol{x}_{s-p+1}$	$\boldsymbol{x}_{s-p+2}$	..	$\boldsymbol{x}_{s-1}$	$\boldsymbol{x}_s$	

Table 4.6: Two layer LSTM data point: subsequence feature matrix and its label.

window of length p. Define the size of $\boldsymbol{\xi}$ as d. Suppose that the time series fold starts at index s. For each time step $t \in (p+1, s-1)$, we create a feature matrix such that y_t is taken as a label, and its corresponding feature matrix with dimension $d \times p$ is formed from the stacked $(\boldsymbol{\xi}_{t-p+1}, \boldsymbol{\xi}_{t-p+2}, .., \boldsymbol{\xi}_{t-1}, \boldsymbol{\xi}_t)$. The 3-dimensional training set is made from these multivariate subsequences, and has the dimension of $(s-p-1) \times d \times p$, where the first dimension specifies the number of subsequences made with a sliding window from the whole time series up to the point s.

The specified procedure can produce a 1-day forward forecasts given that the previous p sales values y and external regressors $\boldsymbol{x}$ are known. However, the required forecast horizon is seven days. Suppose that the forecast starts at time step s, it should end at time $s+6$ inclusive, and the last known sales value is y_{s-1}. We assume that the future external regressors $\boldsymbol{x}_s$ to $\boldsymbol{x}_{s+6}$ are known in advance. In order to produce a multistep forecast, we use the same approach as with the GLM: we substitute the forecast $\hat{y}_s$ for y_s when we form the input feature vectors $\boldsymbol{\xi}_{s+1:s+6}$. For the first time step s, the input vector contains the values $\boldsymbol{\xi}_{s-p+1}$ to $\boldsymbol{\xi}_s$, since the last vector $\boldsymbol{\xi}_s$ is simply a concatenation of the past sales value y_{s-1}, and a vector of external regressors $\mathbf{x}_s$, which is known in advance. For the step $s+1$, the input has to contain all vectors $\boldsymbol{\xi}_{s-p+2}$ to $\boldsymbol{\xi}_{s+1}$. However, in the vector $\boldsymbol{\xi}_{s+1}$, the sales y_s is not known beforehand. Thus, we replace the sales history value y_s with its forecast from the previous step $\hat{y}_s$ to form a vector $\hat{\boldsymbol{\xi}}_{s+1}$, which replaces the vector $\boldsymbol{\xi}_{s+1}$. We continue in this manner until the full required multistep forecast (7 days in our experiments) is collected.

The regularization is important for the presented deep LSTM architecture since this NN tends to overfit for some product-location sales time series within a small number of epochs. The regularization is done as in the case of MLP with both the dropout and early stopping. The dropout is applied to the output of the **LSTM 2** cell before the **Dense** layer. The early stopping is applied to all folds during the hyperparameter selection CV procedure in the same way as with the two-layer MLP neural network in Section 4.3.4.2. Thus, the number of epochs is selected like a hyperparameter. The two-layer LSTM architecture presented here also has a large number of other important hyperparameters which have to be tuned in cross-validation: the

LSTM Hyperparameter	Range/Set
Window size (**ws**)	$\{3, 7, 14, 28\}$
Neurons, **LSTM 1** cell	from $\lfloor 1.5 \times (\mathbf{ws} + 13) \rfloor$ to $\lfloor 2.5 \times (\mathbf{ws} + 13) \rfloor$
Neurons, in the **LSTM 2** cell	from $\lfloor (\mathbf{ws} + 13)/2 \rfloor$ to $\lfloor 1.5 \times (\mathbf{ws} + 13) \rfloor$
Dropout	$\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$
Number of epochs in training	$\{150, 200, 250, 300, 350, 400\}$

Table 4.7: Two layer LSTM hyperparameter ranges used in cross-validation.

sliding window size (number of look-back vectors $\boldsymbol{\xi}_t$), the size of the **LSTM 1** cell, and the size of the **LSTM 2** cell. The hyperparameters for this deep LSTM are conveniently summarized in Table 4.7.

4.3.4.4 Optimization in Neural Networks

Previously we described two heuristics for improving the mini-batch gradient descent procedures: momentum updates (Equation 2.51) and RMSprop updates (Equation 2.53). In our MLP and LSTM implementations we use the MBGD procedure that combines the momentum updates with RMSprop, called Adam. The Adam equations can be presented as follows [51]:

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\Big(\nabla_{\hat{\boldsymbol{\theta}}_{t-1}} \mathcal{L}(\mathbf{X}_m)\Big) \tag{4.36}$$

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\Big(\nabla_{\hat{\boldsymbol{\theta}}_{t-1}} \mathcal{L}(\mathbf{X}_m)\Big)^2 \tag{4.37}$$

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - \beta_1^t} \quad \hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - \beta_2^t} \tag{4.38}$$

$$\hat{\boldsymbol{\theta}}_t = \hat{\boldsymbol{\theta}}_{t-1} - \alpha \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \boldsymbol{\epsilon}} \tag{4.39}$$

where the parameters $\beta_1, \beta_2 \in [0, 1)$ regulate the first and second moment updates, α is the learning rate, and $\boldsymbol{\epsilon}$ is a vector of small constants to prevent the division by zero. The gradient of the loss function computed for the mini-batch $\mathbf{X}_m$ is denoted as $\nabla_{\hat{\boldsymbol{\theta}}_{t-1}} \mathcal{L}(\mathbf{X}_m)$. The subscript of the gradient $\nabla_{\hat{\boldsymbol{\theta}}_{t-1}}$ means that it is evaluated at the point $\hat{\boldsymbol{\theta}}_{t-1}$. The vectors $\boldsymbol{m}_0$ and $\boldsymbol{v}_0$ are conveniently initialized to zero at the beginning of the Adam optimization, which means that initial algorithm updates are biased towards zero. Equations 4.38 are needed to remove this initialization bias.

The authors suggest that Adam initial optimization parameters should be chosen as follows: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\boldsymbol{\epsilon}_i = 10^{-8}$, and $\alpha = 0.001$ [51].

Surprisingly, these initial parameters work very well for the MLP and LSTM architectures introduced before. The most important property of the algorithm is that due to the moment updates $\boldsymbol{v}_t$ and $\boldsymbol{m}_t$, the speed of the gradient descent procedure can increase or decrease even though the learning rate stays the same, which allows to explore the loss function surface with greater precision under fewer restrictions. Due to the first and second moment normalization, the Adam optimization procedure is very robust against the noise and dramatic gradient direction changes introduced by the mini-batches [51]. Since the sales data is very noisy, and the mini-batches are supposed to be very diverse, Adam is the method of choice for the optimization in the MLP and LSTM models presented above. In the literature and in the neural network applications, Adam is considered to be an algorithm of choice for many problems. However, the most recent research suggests that Adam might be inferior to SGD with some state-of-the-art deep learning models in its generalization capability [77]. Based on the preliminary investigation for our sales dataset, the Adam optimization routine outperforms SGD, steepest gradient descent and RMSprop in demand forecasting due to its faster speed of convergence, while demonstrating comparable out-of-sample performance.

In order to train both the MLP and LSTM-based models, we use the mean squared loss function defined in Equation 2.1. For a specific batch of size M, label y, and NN output $\hat{y}$, the mean squared error can be written as

$$\mathcal{L}(\boldsymbol{y}_{1:M}, \hat{\boldsymbol{y}}_{1:M}) = \frac{1}{M}\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 \tag{4.40}$$

We train the two-layer MLP and the two-layer LSTM models with the batches of size 64. All the neural network weights in both architectures are initialized with the help of Glorot uniform sampling, also known as Xavier uniform initialization. This initialization strategy is particularly useful in deep neural networks, since it preserves the gradient variances for initial layers, where the gradients usually diminish due to the backpropagation [36]. In this scheme, the weights of the neurons in layer i are drawn from the uniform distribution

$$\text{Uniform}\Big(-\frac{\sqrt{6}}{\sqrt{n_{i-1}+n_i}}, \frac{\sqrt{6}}{\sqrt{n_{i-1}+n_i}}\Big) \tag{4.41}$$

where n_i is the number of neurons in the layer i. For the **LSTM 1** cell, the number n_{i-1} is equal to the input $\boldsymbol{\xi}_t$ size, while for the **LSTM 2** cell, it equals the size of the **LSTM 1**'s internal state. The number of neurons n_i in both LSTM layers is equal to the size of their respective memory cells. All biases are initialized with zeros.

4.3.4.5 Hyperparameter Tuning

As mentioned in the previous sections, both the deep MLP and deep LSTM have a large number of hyperparameters to tune. The hyperparameters of these models significantly influence their generalization capability in demand forecasting. The most important MLP hyperparameters are presented in Table 4.5. The hyperparameters of the two-layer LSTM model are presented in Table 4.7.

In order to select the hyperparameters, we use the time series cross-validation as discussed in Section 4.2.1. However, the training in neural networks is very slow, and the time series is split into a smaller number of folds during model selection in comparison to the Poisson GLM and dynamic harmonic regression. As described earlier, the validation portion of the time series contains 53 weeks overall. For the deep MLP, each fold in the hyperparameter cross-validation contains 7 weeks which corresponds to 8 folds overall. In the deep LSTM hyperparameter cross-validation each fold contains 11 weeks, which amounts to 5 folds. The model evaluation CV which is run on the test part of the time series remains the same for the neural network models as for the Poisson GLM and dynamic harmonic regression: all the 53 folds are of weekly length.

The number of folds influences the model evaluation and hyperparameter selection. As described above, the neuronal weights in both architectures are initialized using Glorot uniform sampling. Depending on the initialization, the neural network model can produce different results, which can influence the model selection. An obvious solution would be to always start training from the same initial values. However, this approach is discouraged in practice, since when choosing an arbitrary initialization for the neural network, there is no guarantee that it is not a bad starting point. Therefore, evaluating the neural network with multiple parallel weight initializations, and finding the average of these is the golden standard [7]. However, we do not have the computational resources in NN demand forecasting to perform the model training for multiple random seeds. Instead, we initialize new weights with Glorot sampling for each fold. Thus, the bagging of models with random initial weights takes place across folds.

Another problem in neural networks is how the hyperparameter candidates are chosen. Traditional approaches to this problem include the grid search, random search, and coordinate descent [7]. We use the grid search procedure for both the dynamic harmonic regression and Poisson GLM. However, this procedure is too slow to use in our MLP and LSTM architectures, since the number of suggested combinations is too large. In the MLP model, the grid search suggests around 3.25 millions of hyperparameter sets. As for

the LSTM, the number of suggested hyperparameter combinations is around 128 thousands for both types of seasonal modelling. This number of combinations is impossible to test within the cross-validation framework. Another problem of the grid search is that it splits the hyperparameter space into multiple factors, where each factor can be viewed as an axis of the grid. The number of suggested hyperparameter combinations is solely determined by the grid granularity, and specified only through the grid axes. For example, if the required number of the hyperparameter suggestions is a prime number, then the grid search has to be replaced with another procedure. In order to solve these problems, we introduce a procedure that combines the grid search and a low-discrepancy sequence called the Halton sequence.

To explain the concept of low-discrepancy sequences we first introduce the concept of star discrepancy presented by Niederreiter [60]. Given an s-dimensional half-open cube $\mathbb{I}^s = [0,1)^s, s \geq 1$, a sequence of N points $x_1, x_2, .., x_N$, where $x_i \in \mathbb{I}^s$, and a subinterval J of the cube $\mathbb{I}^s$, we can define the discrepancy D as follows:

$$D = \sup_J |\frac{A(J;N)}{N} - V(J)| \tag{4.42}$$

where $A(J;N)$ denotes the number of points x_i in the subinterval J, $V(J)$ is the volume of J, and the supremum is extended over all half-open subintervals $J = \prod_{j=1}^{s}[0, u_j]$ of $\mathbb{I}^s$. The goal of the low-discrepancy sequence is to minimize the star-discrepancy [19]. In simpler terms, the sequence can be considered low-discrepancy, if the proportion of sequence points falling within any region of some space S is roughly equal to the ratio of this region volume to the whole S-space volume. An interesting property of such sequences following from the discrepancy definition is that these sequences seem to be random due to their distribution in space, even though they are completely deterministic. The Halton sequence is one example of a low-discrepancy sequence.

Suppose that we need to generate a sequence of n quasi-random m-dimensional points. We represent each of the m dimensions with a separate prime number. Given a prime number b for one of the dimensions, we can find one-dimensional Halton sequence numbers $h_i, 1 \leq i \leq N$. In order to do this, we express the index i of the number h_i in terms of the prime number b as its base. For example, given $b = 3$, we convert each index into the ternary numerical system where $1 = 1_3, 2 = 2_3, 3 = 10_3, 4 = 11_3$, while for the prime number 7, the index $i = 10$ would be $10 = 13_7$. Next we write the prime-base index representations in reverse order, and put them after the decimal point in the prime-base. For a ternary sequence, it would become $1 \rightarrow 0.1_3, 2 \rightarrow 0.2_3, 3 \rightarrow 0.01_3, 4 \rightarrow 0.11_3$, etc., and for the base 7,

the decimal index $i = 10 \rightarrow 0.31_7$. After all the indices i have been converted into their respective prime-base, and inverted as described above, we rewrite them back in the decimal format, where each index i, converted back to the decimal form, will be the number h_i of the Halton low-discrepancy sequence [19]. Each prime number b has its own Halton sequence. To produce a multidimensional Halton sequence, we simply combine Halton sequences from the different prime numbers.

The main benefit of the Halton sequence is that any number of points can be generated from it, and they will cover the search space evenly. The produced numbers will be in the range from 0 to 1. In the case the hyperparameters have a different range, we simply multiply the number in the Halton sequence by the hyperparameter range and add it to the lowest hyperparameter bound. For example, given a hyperparameter i range from a_i to b_i, and a Halton number $h_i \in [0, 1)$, we transform h_i as follows:

$$h_i' = a + (b - a) \times h_i \tag{4.43}$$

With m hyperparameters, we produce an m-dimensional Halton sequence as described above, and convert each Halton number h_i into its respective hyperparameter range with Equation 4.43. If the hyperparameter is an integer, the generated Halton number is first converted with (4.43), and then rounded to the closest integer with the floor operator.

A feasible alternative to the Halton sequences is the random search. The random search is capable of exploring the hyperparameter space efficiently, often giving better solutions than the manual or grid-search strategies [10]. However, the random search can produce the hyperparameter sets which are clamped to each other as demonstrated in Figure 4.12. The areas outlined in red demonstrate the clamping phenomenon. It is easy to notice, that the Halton sequence of points is much more evenly distributed than that of the random search. If the number of sampled points from the hyperparameter space is very large, then the random search can give comparable or better results than the Halton sequence, since the latter is still a determinate search on a nonlinear grid. However, if the number of sampled hyperparameter sets is small, the random search does not explore the hyperparameter space as well as low-discrepancy sequences do.

For each MLP and each LSTM model separately, the hyperparameter search is conducted using 3 loops: outer, middle, and inner loop. In the outer loop, we try all window sizes one by one. For each window size, an m-dimensional (m equals the number of hyperparameters for the model without the window size and epochs) Halton sequence is generated, which determines the middle loop. 30 points of the Halton sequence are sampled for the both

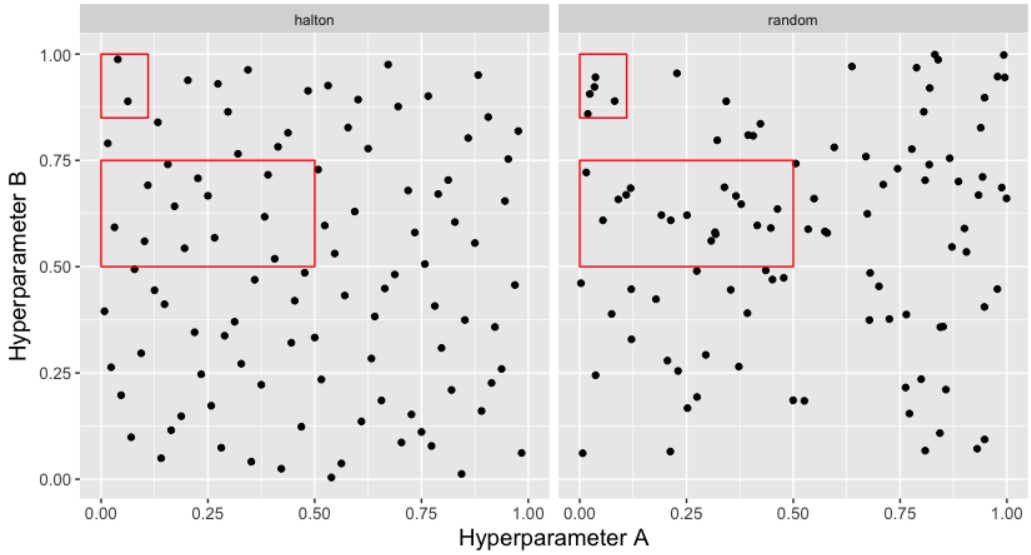

Figure 4.12: The Halton sequence points compared to the points produced by the random search in the hyperparameter space.

MLP models, and 20 Halton sequence points are sampled for the both deep LSTMs. We sample more points for the MLP, since it has more hyperparameters to tune. For each point in the Halton sequence, we run the model cross-validation. The early stopping technique, which iterates through the epochs in each fold in cross-validation as described before, can be viewed as the inner loop.

Chapter 5

Implementation

In order to select and evaluate fast-moving product demand forecasting models, we use a grid of 4 processors, which amounts to 176 threads. We evaluate each model class (baseline naive method, dynamic harmonic regression, Poisson GLM with elastic net, two-layer MLP and two-layer LSTM with Fourier seasonality modelling, two-layer MLP and two-layer LSTM with deseasonalized sales time series) independently in succession. All product-location time series are evaluated in parallel in their own threads.

The computation code is written in R. In order to create the Fourier seasonality terms, we use the **TSA** (periodogram) and **forecast** (sine/cosine term creation) R packages. The weekly seasonality is removed with the function **decompose** from the **stats** package in R. The dynamic harmonic regression models are fit using the **arima** function in the **stats** package. For the Poisson GLM with elastic net, we use the **glmnet**-package in R. The deep learning NN models are created and trained within the **Keras** library interface in R with the **Tensorflow** backend. Halton sequences are produced with the help of **randtoolbox** package.

The overview of the model comparison procedure for demand forecasting is given in Figure 5.1. The sales data is read and preprocessed during the initialization. At this stage, all the hyperparameter ranges are initialized (Section 4.3.2.3 for dynamic harmonic regression, Table 4.4 for the elastic net Poisson GLM, Table 4.5 for the two types of the two-layer MLP, Table 4.7 for the two types of the two-layer LSTM). Next, the initialization procedure creates 100 threads: one for each sales time series. Given a model class, everything related to a particular product-location demand forecasting is done in 1 thread.

The model selection stage involves trying various hyperparameters generated with the grid search or Halton sequences, and selecting the 4 best models for each product-location time series based on MAE, RMSE, MAPE,

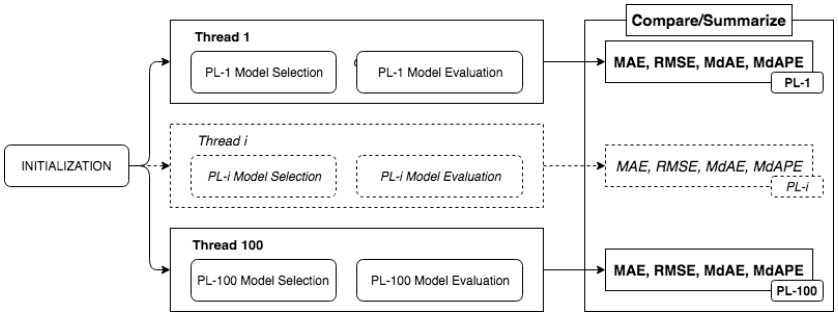

Figure 5.1: Demand forecasting model comparison procedure.

and MdAPE. This stage is performed with the CV on the validation part of the time series. The seasonal naive method does not have any hyperparameters to tune. The model selection procedure can be conveniently summarized with the following steps:

1. **Preprocess the data for all time steps**

   Before the model can be selected, we add the stockout variables as described in Section 4.2.5 to each date, and scale the weather regressors as explained in Section 4.2.3.

2. **Generate a sequence of hyperparameter vectors**

   At this stage, the hyperparameter space Ξ (defined in Section 2.2.2) is created with the grid search (Section 4.3.2.3 for dynamic harmonic regression, Section 4.3.3.2 for Poisson GLM) and Halton sequences (only for MLP- and LSTM-based models, as described in Section 4.3.4.5).

3. **Produce cross-validation folds for model selection**

   For each product-location, split the sales time series validation part into 53 cross-validation folds as described in Section 4.2.1. A list of 53 consecutive fold start indices begins 106 weeks before the end of the times series. These indices are exactly one week apart from each other.

4. **Preprocess the fold data**

   Given a fold start index, model-specific time series preprocessing is done. Log-transformation (Equation 4.5) is done for all models except the Poisson GLM. For the Poisson GLM with elastic net we convert the floating-point sales quantities into integers with Equation 4.6.

5. **Model seasonality**

   Create the seasonal Fourier features as described in Section 4.2.4, or remove the seasonality altogether as described in Section 4.3.4.1.

6. **Train the model**

   A specific model is created from a set of hyperparameters $\boldsymbol{\xi}_i \in \Xi$ supplied to a model class, and trained for each test fold. The trained model produces a 7-day forecast for the folds. Each forecast is converted back to the original scale by undoing transformations done in step 4.

7. **Evaluate the error metrics**

   Given the forecasts from the step 4, evaluate the error metrics MAE, RMSE, MAPE, and MdAPE for all 53 weeks of the time series validation part as described in Section 4.2.2.1. Skip all the days with zero sales as mentioned in Section 4.2.5. For each error metric, check if it is smaller than the same metric evaluated so far for other sets of hyperparameters $\boldsymbol{\xi}_j \in \Xi$, $j < i$. If it is, save the new set of hyperparameters $\boldsymbol{\xi}_i$ as the best one so far.

8. **Select the next set of hyperparameters**

   Select the next set of hyperparameters $\boldsymbol{\xi}_{i+1}$ from the list Ξ until all hyperparameter vectors have been tried. Go back to step 3.

9. **Output the best set of hyperparameters**

   After all the options in the hyperparameter space Ξ have been exhausted, the best hyperparameter sets for the 4 error metrics determine the 4 best models, one for each error metric. The best models for MAE, RMSE, MAPE, and MdAPE are passed to the model evaluation stage.

The model evaluation stage has the same steps as presented above, except that the steps 2 and 8 are skipped. The hyperparameter space is not created at the model evaluation stage: the best set of hyperparameters passed from the model selection stage is used for the single round of cross-validation. In step 3, the folds are created over the last 53 weeks of each product-location time series, unlike the penultimate year in the model selection cross-validation. In the error evaluation step 7, the best vector of hyperparameters is not tracked. The output step 9 of the model evaluation stage produces the 4 error metrics that are finally written to a file for further analysis.

Chapter 6

Experimental Results

The performance of the best models from 7 model classes (dynamic harmonic regression, Poisson GLM with elastic net, two-layer MLP or LSTM with Fourier seasonality modelling or deseasonalization, seasonal naive method) in demand forecasting of 100 fast-moving product-locations is measured with the MAE, RMSE, MAPE, and MdAPE metrics. We further refer to a model class as simply **model**. For each error metric, we summarize the model performance in terms of its rank. Each model has a rank from 1 to 7 for a particular product-location in relation to the other models according to each forecast error metric in the cross-validation. The rank 1 corresponds to the smallest error among all models, while the rank 7 means that the model has the largest error for a given product-location. The final rank of a model for a particular error metric is the average of its ranks across all product-locations. We conveniently summarize these results in Table 6.1

	MAE	**RMSE**	**MAPE**	**MdAPE**
DHR	1.39	1.91	1.66	2.16
GLM	3.92	2.57	5.83	3.94
MLP-F	3.91	3.72	3.66	3.92
MLP-D	3.65	3.92	2.88	3.74
LSTM-F	4.65	4.85	4.21	3.93
LSTM-D	3.52	4.13	3.25	3.35
Baseline	6.96	6.9	6.51	6.96

Table 6.1: Model ranks across 100 product-locations for each error metric.

In Table 6.1, **DHR** stands for dynamic harmonic regression, **GLM** refers to the Poisson GLM with elastic net, and **baseline** means the naive seasonal forecasting model. The 2-layer **MLP** and **LSTM** models that operate on the time series with Fourier seasonality terms have an **F**-suffix in their name, or **MLP-F** and **LSTM-F** respectively. The same models operating on deseasonalized time series have the suffix **D**, and are called respectively **MLP-D** and **LSTM-D**. This naming policy is used further.

In Table 6.1, the **dynamic harmonic regression** shows the best results in terms of all 4 error metrics, while the **baseline** model is consistently the worst one in all metrics which was expected. **GLM** performs very well with respect to RMSE, but is considerably worse than other models for MAE, MAPE and MdAPE. MAPE and MdAPE of each model for all product-locations are presented separately in Figure 6.1 (MAPE) and Figure 6.2 (MdAPE).

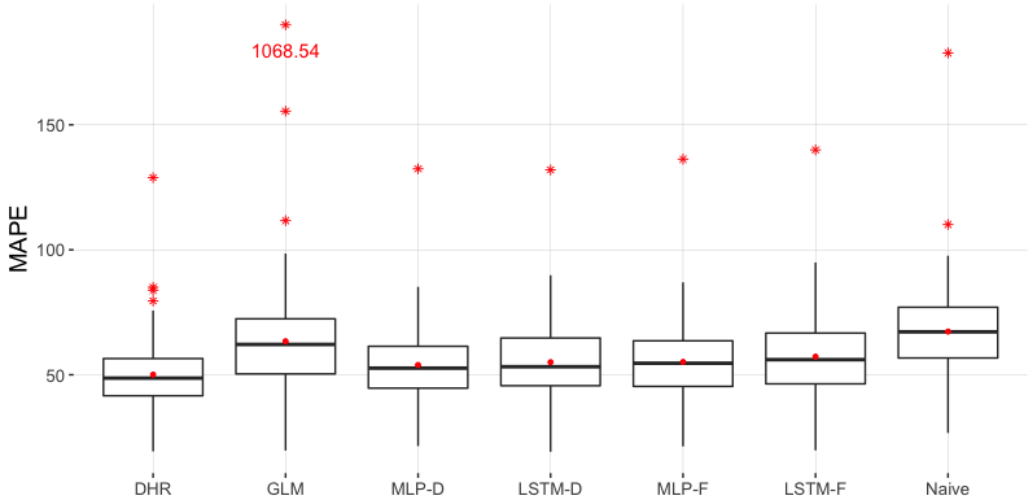

Figure 6.1: MAPE boxplot for each type of model.

The boxplots follow the definition by Tukey [31], where the box spans from the 1st quartile (Q_1) to the 3rd quartile (Q_3). The interquartile range (IQR) is specified as $\text{IQR} = Q_3 - Q_1$. The upper whisker extends from Q_3 to $\min(Q_3 + 1.5 * \text{IQR}, \text{obs}_{\max})$, and the lower whisker ranges from $\max(Q_1 - 1.5 * \text{IQR}, \text{obs}_{\min})$ to Q_1. The points marked with asterisks outside of whiskers represent outliers. In the MAPE boxplot, **GLM** has a very large outlier at 1068%. For convenience, it is labelled in red, and moved to the y coordinate which corresponds to the MAPE of 190%. The mean MAPE and MdAPE of each model are marked with the red dots inside the boxes.

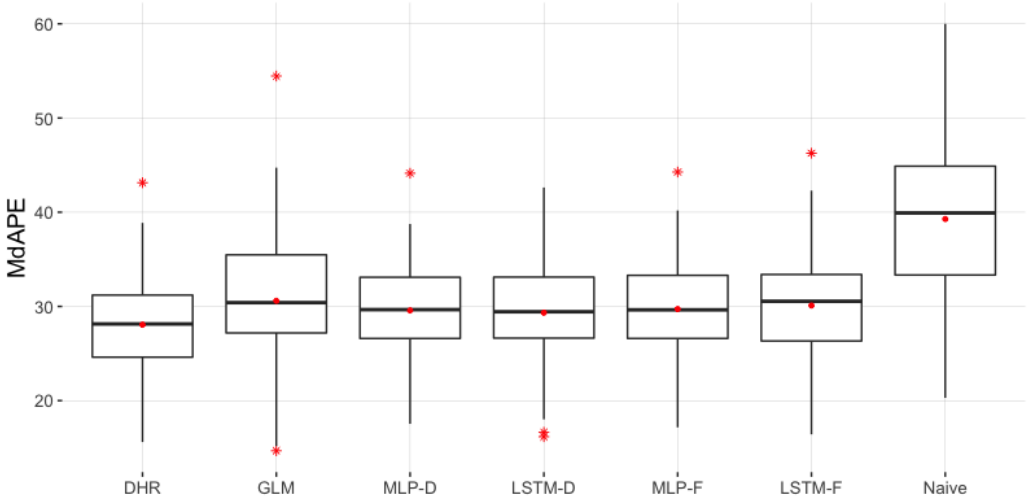

Figure 6.2: MdAPE boxplot for each type of model.

The boxplots indicate that **DHR** seems to perform better than the alternatives. The **baseline** forecasting method is clearly the worst. Neural network models have very close results, and it is not at once clear which one is the best. **GLM** performs the second worst: its 3rd quartile is higher than those of neural networks for MdAPE, while for MAPE, its 1st and 3rd quartiles as well as the median are worse than other models' respective summaries except for **baseline**. Moreover, **GLM** has the largest outliers out of all models.

The full sample of ranks corresponding to all product-locations for all models and error metrics is presented with a bar chart in Figure 6.3. Figure 6.3 is divided into 4 horizontal blocks, where each block represents one error type. Each block has the model categories on the left. For each error metric, a model always has 100 ranks which correspond to 100 product-location sales histories. The ranks of each model are encoded with the color scheme, which allows to easily deduce from the graph how well the model performs across all product-locations with respect to some metric in comparison to other models.

From Graph 6.3, we can notice that the **baseline** model is consistently the worst one across all error metrics: it rarely scores better than the last place. On the other hand, **DHR** is consistently getting the best ranks across all error metrics. It is the only model that never gets beaten by the naive seasonal forecasting method. **GLM** model performs well only in RMSE, and is better than only **LSTM-F** in MAE and MdAPE. **GLM** MAPE is the

Figure 6.3: Full rank distributions for each type of model and error.

worst after the **baseline** model. The latter has better MAPE than **GLM** in 25% of product-location demand forecasts. Surprisingly, the **baseline** method manages to achieve the second rank in MAPE in a few time series. The deseasonalized NN models perform relatively better than their Fourier seasonality counterparts. In MAE and RMSE, **MLP-D**, **LSTM-D**, and **MLP-F** seem to perform equally well. However, deseasonalized models seem to have better ranks than Fourier seasonal models in MAPE and MdAPE. In MAPE, **MLP-D** seems to outperform **LSTM-D** a little, while in MdAPE, **LSTM-D** oftener has better ranks than **MLP-D**. **LSTM-F** has considerably worse ranks than **LSTM-D** and **MLP** models in all error metrics except for MdAPE, where it is comparable to **MLP-D**.

We further investigate the effects of Fourier seasonality modelling against seasonal decomposition in neural networks. In Figure 6.4, we plot the number of product-locations where a particular rank from 1 to 7 for each error metric was achieved either with Fourier seasonal modelling or deseasonalization in **MLP**. A similar bar chart is produced for **LSTM** in Figure 6.5.

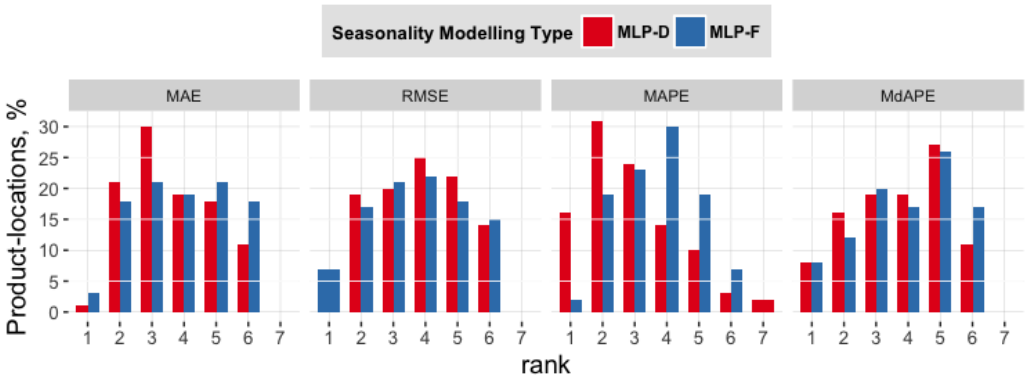

Figure 6.4: MLP rank results for each type of seasonality modelling and error metric.

Seasonal decomposition in neural networks shows generally better results than Fourier seasonality modelling in all error metrics as evidenced by the taller red bars for smaller ranks of the bar charts. In order to summarize this observation more thoroughly, we perform a sign hypothesis test [22] on the rank results for each NN model as described in Section 4.2.2.2. We compare the model **MLP-D** against **MLP-F**, and **LSTM-D** against **LSTM-F**. For each error metric, we use the paired sign test, since the ranks $(r_i^{\mathrm{D}}, r_i^{\mathrm{F}})$ of the two models **D** and **F** are dependent for a product-location i, but independent from the pair $(r_j^{\mathrm{D}}, r_j^{\mathrm{F}})$, $i \neq j$, where **D-F** means **MLP/LSTM-D**

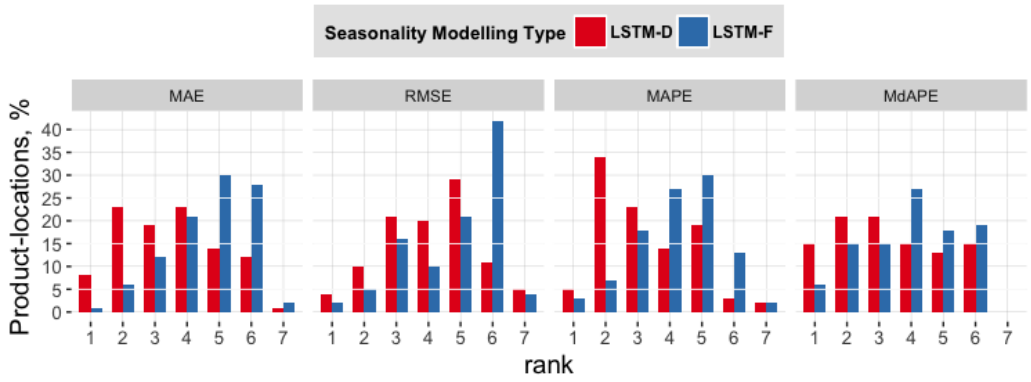

Figure 6.5: LSTM rank results for each type of seasonality modelling and error metric.

against **MLP/LSTM-F** respectively. The null hypothesis is that for a given product-location, the tested error metric is as likely to be lower for the deseasonlized version of the model, as for the Fourier-version of the model. Given our observation of graphs (6.4) and (6.5), we have a strong suspicion that the deseasonlized NN models are better than their Fourier-term counterparts. Thus, the alternative hypothesis is that the NN models working with deseasonalized time series consistently have lower ranks. Due to this prior assumption, we select the right-tailed version of the test, with a significance level of 5%. For any product-location i, we restate the paired sign test for the model ranks as follows:

$$\mathrm{H}_0: \quad \theta_M = P_M(\mathrm{r}_i^{\mathrm{D}} < \mathrm{r}_i^{\mathrm{F}}) = 0.5 \tag{6.1}$$

$$\mathrm{H}_1: \quad \theta_M > 0.5 \tag{6.2}$$

where θ_M is the probability that the model **D** is better than the model **F** for the error metric M, or alternatively the model **D** has a higher probability of achieving a lower rank than the model **F**. We run the paired sign test for all error metrics. The main assumptions are stated in Section 4.2.2.2. The rank paired sign test is the only option to compare MAE and RMSE errors since they are scale dependent, which means the differences are not sampled from the same continuous distribution. Other tests can be used for MAPE and MdAPE. Nevertheless, the use of the t-tests and median comparison with Wilcoxon rank tests with MAPE and MdAPE is precluded by the significant asymmetry of the observed errors, and their paired differences. The p-values for the sign test of **MLP-D** against **MLP-F**, and **LSTM-D** against **LSTM-F** for all error metrics are presented in Table 6.2. In the table, we present the

alternative hypothesis that the first model is better than the second model ($\theta_M > 0.5$) with the $<$ - sign.

	Sign test p-values			
	MAE	**RMSE**	**MAPE**	**MdAPE**
MLP-D < MLP-F	0.30865	0.86437	0.00009	0.18410
LSTM-D < LSTM-F	0.000006	0.0009	0.00002	0.01760
MLP-F < MLP-D	0.75794	0.18410	0.99996	0.86437

Table 6.2: Comparison of Fourier seasonal modelling against deseasonalization: sign test p-values for MLP and LSTM.

From the review of Table 6.2, we assume that the model **LSTM-D** performs consistently better than **LSTM-F** for all error metrics. For **MLP**, the situation is not as clear: only the null-hypothesis for the MAPE error metric is rejected. For MAE, RMSE and MdAPE we keep the null-hypothesis that **MLP-F** is not inferior to **MLP-D**. Next, we reverse the comparison, and test if the model **MLP-F** is better than the model **MLP-D** for MAE, RMSE and MdAPE. According to the p-values, we do not reject the null-hypothesis either. From these tests and Figure 6.4 we know that for MAPE **MLP-D** is better, while in the other metrics it is non-inferior to **MLP-F**. Therefore, a conclusion is made that **MLP-D** is at least as good as **MLP-F**. As for **LSTM-D**, it is strictly superior to its Fourier seasonality counterpart.

We use the fact that MAPE and MdAPE errors are scale-free to find the mean of the model pair differences of these errors for each product-location. The model pairs are (**MLP-D**, **MLP-F**) and (**LSTM-D**, **LSTM-F**). Suppose that the model A produces an error e_i^A, while the model B produces an error e_i^B for the time series i. We define a **difference sample** to be the set of differences $S_{A-B} = \{e_i^A - e_i^B | 1 \leq i \leq 100\}$. The mean of the differences, or the **difference mean** $A - B$ can be computed on the difference sample S_{A-B}. Then the bootstrap is used to construct the confidence bounds for the difference mean as described in Section 2.3.2. The results are presented in Table 6.3. In the table, both 95% confidence intervals for the difference means are produced with 1000 bootstrap samples from the respective difference samples. The minus sign in the row labels specifies the order of subtraction of the paired error observations. Bootstrap assumes that the difference sample observations are i.i.d. This assumption is true, since MAPE and MdAPE metrics are scale-free, and model performance on one time series does not affect its performance on the other time series.

	MAPE			MdAPE		
	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$
MLP-D – MLP-F	-1.19	-1.91	-0.38	-0.15	-0.50	0.18
LSTM-D – LSTM-F	-2.22	-3.31	-1.03	-0.77	-1.17	-0.37

Table 6.3: Bootstrap confidence bounds on the mean of the difference of deseasonalization against Fourier seasonal modelling in MLP and LSTM.

From Table 6.3, we conclude that **MLP-D** is on the average better than **MLP-F** by 1.19% in MAPE. The MdAPE difference mean of **MLP-D** against **MLP-F** has the lower confidence bound of -0.5% and the upper confidence bound if 0.18%, which confirms the results of the sign test, which hints that the models are equally good in this metric. The **LSTM-D** model is better than **LSTM-F** model by 2.22% in MAPE, and by 0.77% in MdAPE. These results suggest that NN models that work with deseasonalized product-location sales time series are generally better, or at least non-inferior to the Fourier seasonal models in the case of MLP. Based on these results, we restrict our attention to **LSTM-D** and **MLP-D** in further discussion.

	Sign test p-values			
	MAE	**RMSE**	**MAPE**	**MdAPE**
DHR < GLM	10^{-12}	0.006	8×10^{-29}	10^{-5}
DHR < MLP-D	3×10^{-24}	6×10^{-12}	8×10^{-9}	2×10^{-9}
DHR < LSTM-D	2×10^{-17}	1×10^{-16}	1×10^{-12}	4×10^{-5}
GLM < MLP-D	0.933	9×10^{-8}	1	0.903
GLM < LSTM-D	0.903	2×10^{-6}	1	0.956
MLP-D < GLM	0.097	0.999	10^{-21}	0.136
LSTM-D < GLM	0.136	0.999	10^{-20}	0.067

Table 6.4: Sign test p-values for DHR, GLM, MLP-D, and LSTM-D in pairwise comparisons for all error metrics.

We run pairwise one-sided paired sign tests on the error metric ranks of 4 models: **DHR**, **GLM**, **MLP-D**, and **LSTM-D**. The results of these paired sign tests are presented in Table 6.4. The **DHR** ranks are compared to the

ranks of all other models first, since we have a suspicion from Figure 6.3 and boxplots (6.1) and (6.2), that **DHR** is superior to all other models for fast-moving product demand forecasting. **GLM** is compared to the neural network models, where the alternative hypothesis is that the deseasonlized NN models are better than **GLM**. Finally, we compare **MLP-D** and **LSTM-D** in both ways (**MLP-D** $<$ **LSTM-D**, and **LSTM-D** $<$ **MLP-D**).

The sign test significance level is 5%. In Table 6.4 the pairwise comparisons of **MLP-D** and **LSTM-D** have been omitted, since none of them reject the null hypothesis for any of the error metrics. From the sign test p-values, we can see that **DHR** is clearly superior to all 3 alternatives in all error metrics, since the null-hypothesis has been rejected at 5% level for all related tests. This result was expected due to the prior investigation of the boxplots and Figure 6.3. As for **GLM**, this model was clearly superior to neural network models in RMSE. The p-value computed for the test statistic was less than 10^{-6} for the both comparison tests. However, both **MLP-D** and **LSTM-D** are better than **GLM** in MAPE with a very small p-value.

Next we compute the difference means in MAPE and MdAPE for different model pairs, and find the confidence intervals for them, as was done before in the comparison of seasonality modelling strategies. The results of these calculations are presented in Table 6.5.

	MAPE			**MdAPE**		
	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$
DHR – GLM	-22.13	-43.41	-10.91	-2.53	-3.38	-1.80
DHR – MLP-D	-3.87	-4.83	-2.85	-1.51	-1.90	-1.16
DHR – LSTM-D	-4.97	-6.32	-3.88	-1.27	-1.77	-0.80
LSTM-D – GLM	-17.15	-37.76	-6.11	-1.26	-1.97	-0.58
LSTM-D – MLP-D	1.10	-0.04	2.33	-0.24	-0.62	0.13
MLP-D – GLM	-18.25	-39.29	-7.10	-1.02	-1.71	-0.30

Table 6.5: Bootstrap confidence bounds on the mean of the difference of DHR, GLM, LSTM-D, and MLP-D compared pairwise.

In the table we clearly see that **DHR** model outperforms all other models by a large margin. In MAPE, **DHR** is better than NN models by more than 3.87%, and better than **GLM** by 22%. In MdAPE, **DHR** consistently outperforms its competitors by more than 1.27% on the average. All **DHR**

difference means have 95% upper confidence bounds less than zero. It implies that its prediction errors in MAPE and MdAPE are highly likely to be lower than the same errors of competing models. Both **LSTM-D** and **MLP-D** outperform **GLM** by at least 17% in MAPE, and 1% in MdAPE on the average. The examination of error means of **LSTM-D** and **MLP-D** does not reveal the superiority of any model. Given the confidence bounds, we hypothesize that **MLP-D** can be 1% better than **LSTM-D** in MAPE metric, while we cannot make any decisive conclusion about the model performance between **LSTM-D** and **MLP-D** in MdAPE due to the difference mean being too close to zero. These and prior results suggest that **DHR** is a superior model for demand forecasting in fast-moving products.

Next, we limit our attention to product-location sales histories with large outliers. Since almost all product-locations have outliers, we find the sales time series with the largest proportion of them. First, we subtract the weekly seasonality from all the time series using the moving averages method introduced before, with the window size of 7. We define the outliers using the interquartile range IQR. A deseasonalized sales observation can be considered an outlier, if it satisfies the condition:

$$y_{out} > Q_3 + 3 \times \text{IQR} \tag{6.3}$$

where Q_3 is the 3rd quartile. This definition is different from the one used in boxplots! We define the outliers only on one side, since the sales are always positive. Next we select the 25 sales histories which have the most points that satisfy Equation 6.3. We investigate how the models **DHR**, **GLM**, **MLP-D**, and **LSTM-D** compare in these unpredictably fluctuating time series. We depict the ranks of these models in highly fluctuating time series relative to each other. These results are presented in Figure 6.6.

From Figure 6.6, it is clear that **DHR** is even better for sales time series with large frequent outliers. It achieves the first rank in more than 75% of tested product-locations (from 25 overall), and the second rank for the rest. It also has the best ranks for MAPE and MdAPE. For RMSE, **GLM** shows the same performance as **DHR**. **GLM** shows the worst performance out of 4 models in MAE, MAPE, and MdAPE. **MLP-D** is considerably better rank-wise than **LSTM-D** in MAPE, while both models show similar performance in terms of MdAPE. **LSTM-D** is the worst model in RMSE, and second worst in MAE. Given these ranks, we conclude that for fast-moving product-locations with many outliers, **DHR** is the best type of model. It is followed by **MLP-D** in terms of performance, since this NN model does well in MAE, MAPE, and MdAPE, while RMSE, where **MLP-D** is the third, is not as important as the other metrics.

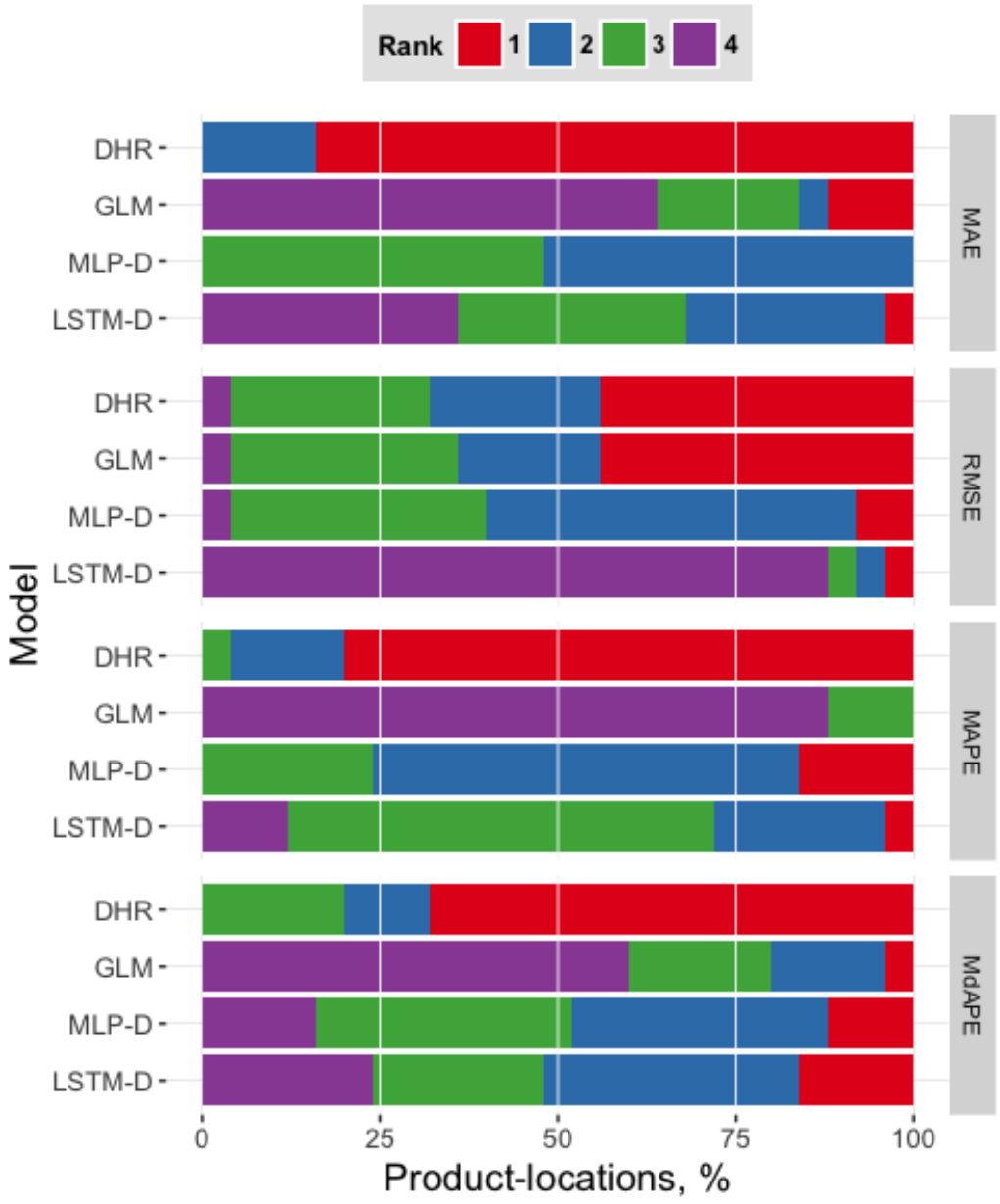

Figure 6.6: Rank distributions of DHR, GLM, MLP-D, and LSTM-D in highly fluctuating time series for each type of error metric.

Next we perform bootstrap sampling to estimate the mean error difference of every pair of the 4 models in terms of MAPE and MdAPE. The bootstrapping is done as before: 1000 samples, and 95% confidence interval on the mean of differences using the percentile method. The results are presented in Table 6.6. From the table, it is evident that **DHR** is better than other models in terms of MAPE by at least 3.5%, and 1.6% in terms of MdAPE. The confidence interval for the mean of differences is always negative which indicates a high certainty about the superior performance of **DHR** in sales time series with large fluctuations. **MLP-D** is clearly better than **LSTM-D** in terms of MAPE, outperforming it by 4.5%. Both neural network models are equal in terms of their MdAPE metric results in highly fluctuating time series. NN models outperform **GLM** by at least 9% in MAPE, and 3% in MdAPE for this subset of sales histories.

	MAPE			**MdAPE**		
	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$
DHR – GLM	-17.96	-22.21	-14.51	-4.67	-6.56	-2.87
DHR – MLP-D	-3.56	-5.16	-2.01	-1.68	-2.42	-0.88
DHR – LSTM-D	-8.16	-11.49	-5.61	-1.60	-2.45	-0.70
LSTM-D – GLM	-9.80	-14.73	-3.96	-3.07	-4.96	-1.32
LSTM-D – MLP-D	4.59	1.51	8.37	-0.08	-0.97	0.83
MLP-D – GLM	-14.39	-17.29	-11.40	-2.99	-4.84	-1.52

Table 6.6: Bootstrap confidence bounds on the mean of the difference of DHR, GLM, LSTM-D, and MLP-D errors compared pairwise in product-locations with high fluctuation.

Next, we examine only the product-location sales time series with the most stable demand. The seasonalities of the sales history are removed with the moving averages as before. We find a deseasonalized sales value y_{obs} stable, if it satisfies the following inequality

$$y_{obs} < Q_3 + 3 \times \text{IQR} \tag{6.4}$$

where IQR is the interquartile range, and Q_3 is the third quartile. We select 25 product-locations with the highest number of sales observations that adhere to Equation 6.4. We compare the ranks of **DHR**, **GLM**, **MLP-D**, and **LSTM-D** on this subset of product-location sales histories. The model

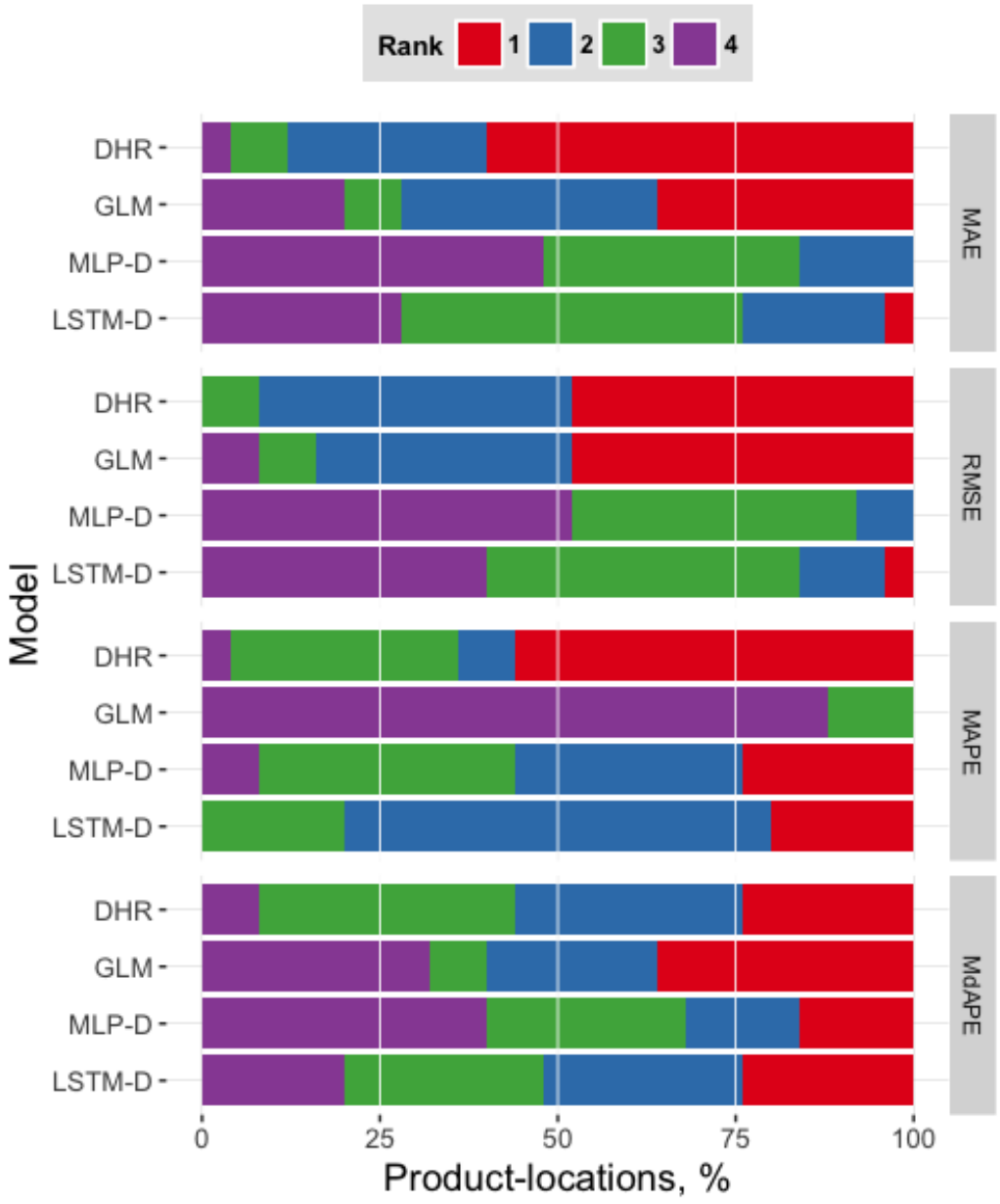

Figure 6.7: Rank distributions of DHR, GLM, MLP-D, and LSTM-D in the 25 most stable time series for each type of error metric.

ranks are presented in Figure 6.7. From the figure, we can notice that **DHR** is better than other models in terms of MAE, and MAPE. However, **GLM** is at least as good as **DHR** in terms RMSE, and might be better that **DHR** in terms of MdAPE. **GLM** is the worst model in terms of MAPE. Interestingly, **LSTM-D** is better than **MLP-D** for MAE, MAPE and MdAPE. Both neural network models perform almost the same rank-wise in RMSE. We perform 1000 bootstrap samples for all pairs of models to find how much the model performance differs in terms of MAPE and MdAPE error difference means. The results are presented in Table 6.7.

	MAPE			**MdAPE**		
	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$	$\hat{\mu}$	$\hat{\mu}_{2.5\%}$	$\hat{\mu}_{97.5\%}$
DHR – GLM	-8.44	-11.36	-5.58	-0.51044	-1.38	0.23
DHR – MLP-D	-2.98	-5.73	-0.59	-1.17	-1.83	-0.48
DHR – LSTM-D	-2.61	-4.95	-0.30	-0.26	-0.93	0.40
LSTM-D – GLM	-5.83	-7.29	-4.39	-0.24	-1.20	0.64
LSTM-D – MLP-D	-0.37	-1.33	0.67	-0.90	-1.51	-0.28
MLP-D – GLM	-5.45	-7.04	-4.01	0.66	-0.28	1.66

Table 6.7: Bootstrap confidence bounds on the difference mean of DHR, GLM, LSTM-D, and MLP-D errors compared pairwise in stable demand product-locations.

From Table 6.7 we notice that **DHR** is still vastly superior to other models in terms of MAPE. However, for stable-demand sales time series, its MdAPE is similar to that of **GLM** and **LSTM-D**. Both NN models outperform **GLM** in terms of MAPE, but are not superior to it in MdAPE. Surprisingly, **GLM** is the second best model in MdAPE for stable demand after **DHR**. In stable demand time series, **LSTM-D** is superior to **MLP-D** in MdAPE by 0.9%, and has the same performance in MAPE.

Chapter 7

Discussion

The results demonstrated in Chapter 6 showed that some advanced statistical methods such as dynamic harmonic regression are non-inferior to neural network models, such as MLP and LSTM with multiple layers, for multi-step demand forecasting of fast-moving products on a daily level. NN models are often selected as a default model to forecast complex data, like time series, due to their capacity to approximate and forecast any possible pattern given a sufficient number of neurons. However, this fact does not specify how the training should proceed in order to learn the desired signal. Neural networks are complex models, that require careful tuning to perform well, and even with a good procedure to select necessary hyperparameters, there are no guarantees that the best underlying hypothesis will be learned.

Our results with respect to NNs and dynamic harmonic regression are in line with the results presented by Arunraj and Ahrens [6], who used SARIMAX and SARIMA-QR to forecast a single fast-moving product demand and compared it to MLP. The authors reported that the linear time series models with external regressors show better results in MAPE, while NNs are better in RMSE. Our experimental results show that indeed **DHR** is a little worse in RMSE relative to its own performance in other error metrics, where it does equally well with **GLM**. In highly stable time series, **DHR** also has much closer results in MdAPE with neural network models. At the same time, our results directly contradict the results presented by Alon [4], who stated that ANNs outperform linear models for any horizon. However, their work used monthly forecasts which have much less noise, and the authors did not explore more advanced linear models like dynamic harmonic regression.

In the experiments, we established that neural networks usually perform better in terms of deseasonlized time series, instead of seasonalities encoded by the Fourier terms. This is confirmed in **LSTM**-based models, while for **MLP**-based models, it is not as clear. However, **MLP-D** is clearly better

than **MLP-F** in MAPE, and non-inferior in other error metrics. In this respect, our results coincide with the results of Chu and Zhang [15], who mentioned that NN models work better with deseasonlized time series than with Fourier-term encoded seasonalities. The authors mention that feature engineering is very important for NN models, which seems to be true given our results, and superiority of deseasonlized NN models over their Fourier counterparts. However, Chu and Zhang arrive at a conclusion that nonlinear models like ANNs are universally better for retail sales forecasting, which our results contradict. The authors had their data aggregated on a monthly level, which leads to sales time series with much less fluctuation than in our experiments. Additionally, they did not assess the models in the presence of external regressors.

One interesting result is that even though the most sales histories in the dataset are count time series, the count-based model (**GLM**), introduced specifically to tackle them, showed also the worst results after the **Naive seasonal** forecasting model. This was especially pronounced in highly volatile time series, where the difference in MAPE between the best model and **GLM** was around 17% (versus 22% for the whole dataset), while for MdAPE this difference was 4.6%. Moreover, for the highly volatile sales histories the model consistently receives the worst metric ranks in MAE, MAPE, and MdAPE. This indicates that **GLM** should not be used for forecasting demand in time series with a lot of unaccounted variation and/or outliers.

At the same time **GLM** did very well in terms of RMSE metric, where it consistently ended up to be the second best model. For the stable demand time series, **GLM** was as good as **DHR** in terms of RMSE. The model performance in terms of MAE is also good for stable demand time series. However, this result is contradicting the performance of **GLM** in MAPE, where this model is consistently the worst one. The probable reason for this is that **GLM** often produces asymmetrically larger forecasts, which result in a large **GLM** error in this metric.

The comparison of neural networks beyond seasonality modelling is not simple, since these models perform almost equally well. Using all the data, **MLP-D** and **LSTM-D** models have almost the same ranks in MAE, RMSE, MAPE, and MdAPE, which is indicated by no paired sign test hypotheses rejected in their rank comparisons. Comparing their MAPE and MdAPE indicates slight superiority of **MLP-D** over **LSTM-D** in MAPE, and maybe a slight advantage of **LSTM-D** over **MLP-D** in MdAPE. However, these difference mean estimates have zeros falling within their confidence intervals, so no definite conclusion can be made in terms of all fast-moving sales time series.

Then we examine the performance of the NN models for the subset of sales

time series with the highest fluctuation. In this subset, **MLP-D** is clearly better than **LSTM-D** for MAE, RMSE, and MAPE. The MAPE is estimated to be around 4.5% better with 95% confidence interval completely below zero. For MdAPE the results are inconclusive. Thus, we conclude that **MLP-D** should be preferred to **LSTM-D** in sales time series with high fluctuation. However, the situations is completely reversed in sales time series with stable demand. In these time series, **LSTM-D** clearly outperforms **MLP-D** in all error metric ranks as indicated by Figure 6.7. The mean of MdAPE differences between the two models in stable sales shows that **LSTM-D** is better than **MLP-D** by 0.9% with the good negative confidence interval. For MAPE, the confidence interval does not provide any evidence of **LSTM-D** superiority. It is concluded that if the sales history does not have many outliers, **LSTM-D** is superior to **MLP-D** for fast-moving product demand forecasting.

The experiments suggest, that linear time series models are a viable approach to time series forecasting. Given rich sales histories of fast-moving products, dynamic harmonic regression provides very good forecasts in situations where sales time series exhibit a lot of fluctuation, while the model performance is still as good as NN model performance in product-locations with the most stable demand. These results suggest that NN models are sensitive to noise, even when thorough regularization is applied. Count-based Poisson regression with elastic net performs best when used in fast-moving product-locations with stable demand, where it shows very good results in MAE, MdAPE, and RMSE. However, its performance severely deteriorates in the presence of outliers. It is unclear why **GLM** performs very well in terms of RMSE, but is the worst in MAPE across all groups of fast-moving product-locations. This can possibly happen due to the asymmetry of MAPE, or some non-linear dependence of variance and mean in the data, which should be further investigated.

The above results are in contradiction with many research papers, and there are multiple reasons why the presented results are observed. A large amount of research is concentrated on univariate demand forecasting, while our work concentrates on time series with covariates. Moreover, our forecasts are multistep, and the models are trained on a daily level. Naturally, it is very different from the monthly or weekly aggregated data used in many demand forecasting experiments, which usually has much fewer fluctuations. We have also noticed a recent heavy bias in literature towards the use of ANN models, where the most recent results often compare neural networks in time series forecasting to other neural networks, or simplistic linear models without proper investigation of complex linear alternatives, such as dynamic harmonic regression or advanced count-time series models.

Chapter 8

Conclusions

In this work, we described six models for fast-moving product-location demand forecasting: dynamic harmonic regression, Poisson GLM with elastic net, two-layer MLP and two-layer LSTM with Fourier seasonality modelling, and the same NN models working with the deseasonalized sales histories. These models were compared to each other in terms of the best multistep daily demand forecasts. We showed how some specific issues in the sales time series such as multiple seasonality and stockouts could be handled. In the process, we introduced a Fourier seasonality modelling approach, and compared it to forecasting the deseasonalized sales time series in ANN models.

We also devoted some space to the description of the theory, the assumptions, and the implementation of the compared models. We outlined the most important details, and pitfalls that could be encountered during the training and prediction in the suggested models. A lot of attention was paid to the optimization methods in dynamic harmonic regression, Poisson GLM with elastic net, and neural networks.

In this work, we outlined the whole process of model comparison for a large number of sales time series. We used a 2-stage cross-validation, where the model hyperparameters were selected in the first CV stage, and the models with the best hyperparameters were evaluated in the second CV stage. We outlined the most important hyperparameters for all models, and the procedures to select them. A new approach to model selection based on Halton sequences was used in the MLP and LSTM models.

We presented the final results with the help of rank comparisons, bootstrap confidence interval estimation, and hypothesis tests. We thoroughly explained to the reader the results of our experimental part, which suggested that advanced linear time series models, such as dynamic harmonic regression, could be a viable alternative to neural networks in fast-moving product demand forecasting.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[2] Aburto, L., and Weber, R. Improved supply chain management based on hybrid demand forecasts. *Applied Soft Computing 7*, 1 (2007), 136–144.

[3] Ali, Ö. G., Sayin, S., Van Woensel, T., and Fransoo, J. Sku demand forecasting in the presence of promotions. *Expert Systems with Applications 36*, 10 (2009), 12340–12348.

[4] Alon, I., Qi, M., and Sadowski, R. J. Forecasting aggregate retail sales:: a comparison of artificial neural networks and traditional methods. *Journal of Retailing and Consumer Services 8*, 3 (2001), 147–156.

[5] Armstrong, J. S., and Collopy, F. Error measures for generalizing about forecasting methods: Empirical comparisons. *International journal of forecasting 8*, 1 (1992), 69–80.

[6] Arunraj, N. S., and Ahrens, D. A hybrid seasonal autoregressive integrated moving average and quantile regression for daily food sales forecasting. *International Journal of Production Economics 170* (2015), 321–335.

[7] Bengio, Y. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

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

[9] Bergmeir, C., Hyndman, R. J., and Koo, B. A note on the validity of cross-validation for evaluating autoregressive time series prediction. *Computational Statistics & Data Analysis 120* (2018), 70–83.

[10] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13*, Feb (2012), 281–305.

[11] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[12] Breiman, L. Statistical modeling: The two cultures. *Quality control and applied statistics 48*, 1 (2003), 81–82.

[13] Brockwell, P. J., Davis, R. A., and Calder, M. V. *Introduction to time series and forecasting*, vol. 2. Springer, 2002.

[14] Chatfield, C., et al. Apples, oranges and mean square error. *International Journal of Forecasting 4*, 4 (1988), 515–518.

[15] Chu, C.-W., and Zhang, G. P. A comparative study of linear and nonlinear models for aggregate retail sales forecasting. *International Journal of production economics 86*, 3 (2003), 217–231.

[16] Cooley, J. W., and Tukey, J. W. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation 19*, 90 (1965), 297–301.

[17] Cox, D. R. Some remarks on overdispersion. *Biometrika 70*, 1 (1983), 269–274.

[18] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems 2*, 4 (1989), 303–314.

[19] Dalal, I. L., Stefan, D., and Harwayne-Gidansky, J. Low discrepancy sequences for monte carlo simulations on reconfigurable platforms. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on* (2008), IEEE, pp. 108–113.

[20] Dalrymple, D. J. Sales forecasting practices: Results from a united states survey. *International Journal of Forecasting 3*, 3-4 (1987), 379–391.

[21] Dickey, D. A., and Fuller, W. A. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American statistical association 74*, 366a (1979), 427–431.

[22] Dixon, W. J., and Mood, A. M. The statistical sign test. *Journal of the American Statistical Association 41*, 236 (1946), 557–566.

[23] Durbin, J., and Koopman, S. J. *Time series analysis by state space methods*, vol. 38. Oxford University Press, 2012.

[24] Dyke, P. P. *An introduction to Laplace transforms and Fourier series*. Springer, 2014.

[25] Ehrenthal, J., Honhon, D., and Van Woensel, T. Demand seasonality in retail inventory management. *European Journal of Operational Research 238*, 2 (2014), 527–539.

[26] Fildes, R., and Hastings, R. The organization and improvement of market forecasting. *Journal of the Operational Research Society 45*, 1 (1994), 1–16.

[27] Flunkert, V., Salinas, D., and Gasthaus, J. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *arXiv preprint arXiv:1704.04110* (2017).

[28] Fox, J. *Applied regression analysis and generalized linear models*. Sage Publications, 2015.

[29] Friedman, J., Hastie, T., and Tibshirani, R. *The elements of statistical learning*, vol. 1. Springer series in statistics New York, NY, USA:, 2001.

[30] Friedman, J., Hastie, T., and Tibshirani, R. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software 33*, 1 (2010), 1.

[31] Frigge, M., Hoaglin, D. C., and Iglewicz, B. Some implementations of the boxplot. *The American Statistician 43*, 1 (1989), 50–54.

[32] Gamberini, R., Lolli, F., Rimini, B., and Sgarbossa, F. Forecasting of sporadic demand patterns with seasonality and trend components: an empirical comparison between holt-winters and (s) arima methods. *Mathematical Problems in Engineering 2010* (2010).

[33] Gardner, G., Harvey, A. C., and Phillips, G. D. Algorithm as 154: An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of kalman filtering. *Journal of the Royal Statistical Society. Series C (Applied Statistics) 29*, 3 (1980), 311–322.

[34] Gelman, A., Stern, H. S., Carlin, J. B., Dunson, D. B., Vehtari, A., and Rubin, D. B. *Bayesian data analysis*. Chapman and Hall/CRC, 2013.

[35] Geurts, M. D., and Kelly, J. P. Forecasting retail sales using alternative models. *International Journal of Forecasting 2*, 3 (1986), 261–272.

[36] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), pp. 249–256.

[37] Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (2011), pp. 315–323.

[38] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. *Deep learning*, vol. 1. MIT press Cambridge, 2016.

[39] Goodwin, P., and Lawton, R. On the asymmetry of the symmetric mape. *International journal of forecasting 15*, 4 (1999), 405–408.

[40] Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., and Schmidhuber, J. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence 31*, 5 (2009), 855–868.

[41] Hamilton, J. D. *Time series analysis*, vol. 2. Princeton university press Princeton, NJ, 1994.

[42] Hansen, J. V., McDonald, J. B., and Nelson, R. D. Some evidence on forecasting time-series with support vector machines. *Journal of the Operational Research Society 57*, 9 (2006), 1053–1063.

[43] Hastie, T., and Qian, J. Glmnet vignette. *Retrieved June 9*, 2016 (2014), 1–30.

[44] Hermans, M., and Schrauwen, B. Training and analysing deep recurrent neural networks. In *Advances in neural information processing systems* (2013), pp. 190–198.

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

[46] Horowitz, J. L. The bootstrap. In *Handbook of econometrics*, vol. 5. Elsevier, 2001, pp. 3159–3228.

[47] Hyndman, R. J. Facts and fallacies of the aic. `https://robjhyndman.com/hyndsight/aic/`, 2013.

[48] Hyndman, R. J., and Athanasopoulos, G. *Forecasting: principles and practice*. OTexts, 2018.

[49] Hyndman, R. J., and Koehler, A. B. Another look at measures of forecast accuracy. *International journal of forecasting 22*, 4 (2006), 679–688.

[50] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[51] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[52] Kohavi, R., et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai* (1995), vol. 14, Montreal, Canada, pp. 1137–1145.

[53] Koza, J. R., Bennett, F. H., Andre, D., and Keane, M. A. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design'96*. Springer, 1996, pp. 151–170.

[54] Kwiatkowski, D., Phillips, P. C., Schmidt, P., and Shin, Y. Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root? *Journal of econometrics 54*, 1-3 (1992), 159–178.

[55] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436.

[56] Lee, M., and Hamzah, N. Calendar variation model based on arimax for forecasting sales data with ramadhan effect. In *Proceedings of the Regional Conference on Statistical Sciences* (2010), pp. 349–361.

[57] Li, F.-F., Karpathy, A., and Johnson, J. Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/neural-networks-1/`. Accessed: 2019-02-08.

[58] Miller, T., and Liberatore, M. Seasonal exponential smoothing with damped trends: An application for production planning. *International Journal of Forecasting 9*, 4 (1993), 509–515.

[59] Nelder, J. A., and Wedderburn, R. W. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General) 135*, 3 (1972), 370–384.

[60] Niederreiter, H. Low-discrepancy and low-dispersion sequences. *Journal of number theory 30*, 1 (1988), 51–70.

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

[62] Pfeffermann, D., and Allon, J. Multivariate exponential smoothing: Method and practice. *International Journal of Forecasting 5*, 1 (1989), 83–98.

[63] Phillips, P. C., and Perron, P. Testing for a unit root in time series regression. *Biometrika 75*, 2 (1988), 335–346.

[64] Rosenblatt, F. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Tech. rep., CORNELL AERONAUTICAL LAB INC BUFFALO NY, 1961.

[65] Ruder, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[66] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533.

[67] Sanders, N. R., and Manrodt, K. B. Forecasting practices in us corporations: survey results. *Interfaces 24*, 2 (1994), 92–100.

[68] Sonoda, S., and Murata, N. Neural network with unbounded activation functions is universal approximator. *arXiv preprint arXiv:1505.03654* (2015).

[69] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[70] Stevenson, W. J. *Operations management*. McGraw-hill, 2005.

[71] Syntetos, A. A., Boylan, J. E., and Disney, S. M. Forecasting for inventory planning: a 50-year review. *Journal of the Operational Research Society 60*, sup1 (2009), S149–S160.

[72] Taylor, J. W. Short-term electricity demand forecasting using double seasonal exponential smoothing. *Journal of the Operational Research Society 54*, 8 (2003), 799–805.

[73] Wasserman, L. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013.

[74] Wedderburn, R. W. Quasi-likelihood functions, generalized linear models, and the gauss-newton method. *Biometrika 61*, 3 (1974), 439–447.

[75] Wen, R., Torkkola, K., Narayanaswamy, B., and Madeka, D. A multi-horizon quantile recurrent forecaster. *arXiv preprint arXiv:1711.11053* (2017).

[76] Williams, P. M. Modelling seasonality and trends in daily rainfall data. In *Advances in neural information processing systems* (1998), pp. 985–991.

[77] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems* (2017), pp. 4148–4158.

[78] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[79] Xu, B., Wang, N., Chen, T., and Li, M. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853* (2015).

[80] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. Apache spark: a unified engine for big data processing. *Communications of the ACM 59*, 11 (2016), 56–65.

[81] Zhang, G. P., and Qi, M. Neural network forecasting for seasonal and trend time series. *European journal of operational research 160*, 2 (2005), 501–514.

[82] Zotteri, G., Kalchschmidt, M., and Caniato, F. The impact of aggregation level on forecasting performance. *International Journal of Production Economics 93* (2005), 479–491.

[83] Zou, H., and Hastie, T. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 67*, 2 (2005), 301–320.