The Application of Machine Learning Methods to Time Series Forecasting

Improving Forecasting Techniques for Smart City Planning in New York City

Authors:

Manuel Alexander Schreiber (676991)

Conor Hasselgaard Cavanaugh (82502)

Supervisor: Prof. Lisbeth La Cour, PhD
(Department of Economics)

Co-Supervisor: Prof. Raghava Rao Mukkamala, PhD
(Department of Digitalization
and Centre for Business Data Analytics)

A thesis presented in partial fullfillment
of the requirements for the degrees of

Master of Science (M.Sc.)
in Advanced Economics and Finance

Cand.Merc.Mat (M.Sc.)
in Erhvervsøkonomi og Matematik

Number of pages: 120
Character count (per page): 226,220 (1,885)
Copenhagen Business School
Copenhagen, September 15, 2020

The Application of Machine Learning Methods to Time Series Forecasting

Improving Forecasting Techniques for Smart City Planning in New York City*

Manuel Alexander Schreiber† and Conor Hasselgaard Cavanaugh‡
Copenhagen Business School (CBS)†,‡
September 15, 2020

Abstract

Smart Cities strive to leverage information and communication technologies in order to analyze the growing amount of available data on the ecosystems of modern cities with the aim of making central infrastructure components and services of a city more interconnected and efficient. A key element of this is knowledge about future conditions which can be obtained through forecasting models. The most recent developments seen in the M4 and M5 forecasting competitions illustrate how far Machine Learning methods have evolved. Therefore, this paper compares and contrasts established statistical forecasting models, ensemble methods as well as recurrent neural networks with respect to point forecasts and prediction intervals. Using publicly available high-frequency data from the New York City Open Data platform, we analyze large univariate time series of traffic flows and Emergency Medical Services (EMS) data. We detect multiple levels of seasonality in the high-frequency data considered in this paper, which is common and which poses special demands to finding suitable forecasting models. We find that recurrent neural networks produce the most accurate point forecasts and uncertainty measures for traffic data. For the EMS data, our results show that both recurrent neural networks and Exponential Smoothing state space models which can account for complex seasonal patterns perform best with respect to point forecasts and uncertainty measures.

Keywords: Bagged ETS, GRU, LSTM, Prediction Intervals, Smart City, TBATS, LGBM, Time Series Forecasting, Traffic Forecasting, EMS Demand Forecasting.

JEL classification: C22, C45, C53.

*As of next year, please refer to the following email addresses if there are any questions concerning our research: *manuelalex.schreiber@gmail.com*, *conorhcavanaugh@gmail.com*. For grading purposes, our code is available here: https://github.com/manu675/MScThesis-Conor-Manu.

†Manuel Alexander Schreiber is a MSc candidate in Advanced Economics and Finance at Copenhagen Business School, Solbjerg Plads 3, DK-2000 Frederiksberg, Denmark
(Email: *masc17aj@student.cbs.dk*).

‡Conor Hasselgaard Cavanaugh is a MSc candidate in Economics and Mangement Science (Cand. Merc. Mat) at Copenhagen Business School, Solbjerg Plads 3, DK-2000 Frederiksberg, Denmark
(Email: *coca13ab@student.cbs.dk*).

Contents

List of Figures

List of Tables

List of Symbols

$\zeta(B)$	Autoregressive polynomial
$\hat{f}^{bag}(\boldsymbol{X})$	Bootstrap Aggregating (Bagging) predictor
$\hat{f}^{boost}(\boldsymbol{X})$	Gradient tree boosting predictor
$\hat{f}_b(\boldsymbol{X})$	Prediction from the b-th bootstrap sample
$1-\alpha$	Nominal confidence level for prediction intervals
δ_j	Backpropagation error of hidden neuron j
η	Learning rate used in the gradient descent algorithm to train a neural network ($0 \leq \eta \leq 1$)
$\mu_{T+h\|T}$	Forecast mean for an ETS model h step ahead forecast
$\nu_{T+h\|T}$	Forecast variance for an ETS model h step ahead forecast
$\mathcal{X}$	Feature space
γ_{jm}	Optimal constant per terminal decision region of the predictor space
$\gamma(PICP)$	Binary variable which is equal to 1 if the PICP is smaller than the nominal confidence level and 0 otherwise
$\nabla J(\boldsymbol{W})$	Gradient of the loss function, which is the vector of the first partial derivatives with respect to the weights of the neural network
$\odot$	Hadamar product or element-wise multiplication
$\psi(B)$	Moving average polynomial
ω	Box-Cox transformation parameter
$\mathcal{Y}$	Output space
ϕ	Dampening parameter ($0 \leq \phi \leq 1$) for the growth term in an exponential smoothing model
$\rho_\alpha(y, \hat{y})$	Pinball loss function with pre-specified significance level α
$\mathbb{Z}^+$	Set of positive integers
$q_\alpha(x)$	α-th conditional quantile function, where $alpha$ is the pre-specified significance level

$\mathbb{R}$	Set of real numbers
$\hat{f}^{rf}(x)$	Random forest predictor
$\hat{\rho}_\epsilon(k)$	Sample Autocorrelation of the model residuals at lag k
$Z_P(B^s)$	Seasonal autoregressive component of order P
Δ_s^D	D-th seasonal difference
$\Psi_Q(B^s)$	Seasonal moving average component of order Q
$\boldsymbol{\theta}$	Main parameter vector for maximum likelihood estimation. Contains e.g. the smoothing and dampening parameters in ETS models and the weights and bias parameters in neural networks
$\boldsymbol{\vartheta}$	Main parameter vector in the TBATS model
ξ	Hyperparameter of the CWC measure to control the violation of coverage probabilities

List of Acronyms

ADAM	Adaptive moments algorithm
ADF	Augmented Dickey Fuller test
AICc	Akaike information criterion with small sample correction
API	Application programming interface
ARCH	Autoregressive conditonal heteroskedasticity
ARIMA	Autoregressive Integrated Moving Average model
ARMA	Autoregressive Moving Average Model
BATS	Exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal components
BPTT	Backpropagation through time
CV	Cross Validation
DCT	Department of City Planning
DGRU	Deep Gated Recurrent Unit neural network
DLSTM	Deep Long Short Term Memory neural network
DoITT	Department of Information Technology and Telecommunications
EMS	Emergency Medical Services
ETS	Exponential Smoothing Models
FDNY	Fire Department of New York City
FFNN	Feedforward neural network
GRU	Gated Recurrent Unit neural network
ICT	Information and Communication Technologies

ITS	Intelligent Transportation Systems
LASSO	Least absolute shrinkage and selection operator
LGBM	Light gradient boosting machine
LSTM	Long Short Term Memory neural network
MAE	Mean absolute error
MAPE	Mean absoute percentage error
MASE	Mean absoute scaled error
MBB	Moving block bootstrap
MODA	Mayor's Office of Data Analytics
MPIW	Mean Prediction Interval Width
MSE	Mean squared error
MTA	Metropolitan Transportation Authority
PI	Prediction interval
PICP	Prediction Interval Coverage Probability
RELU	Rectified linear unit
RMSE	Root mean squared error
RNN	Recurrent Neural Network
RSS	Residual sum of squares
SARIMA	Seasonal Autoregressive Integrated Moving Average model
SE	Standard error
TBATS	Exponential smoothing state space model with trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend and Seasonal components
XGB	Extreme gradient boosting

Acknowledgements

Firstly, we would like to thank CBS for providing us with a top-notch and tuition-free education and for sending both of us on exchange to the University of Texas at Austin, where this collaboration emerged and where we were able to attend Business Data Science classes which have provided us with the necessary background knowledge and programming expertise. Both of those things have been essential in writing our joint thesis on this particular topic. Beyond that, it is fair to say that our time in Texas was a truly unforgettable experience in a social, cultural and academic way.

Next, we would like to thank our supervisor Prof. Lisbeth La Cour as well as our co-supervisor Prof. Raghava Rao Mukkamala for their adhoc and precise support in response to methodological questions. A big thank you also goes out to all those helpful people on Stackoverflow, who spend their free time helping other people with resolving their programming or typesetting software related problems.

Lastly, we would like to thank our parents for financially and personally supporting our long way of higher education.

- Copenhagen, in the Spring of 2020
Manuel and Conor

Section 1

Introduction

Forecasting takes up a major role in corporations with respect to product demand planning, revenue predictions as well as inventory resource planning. All major technology companies including Google, Amazon and Uber now apply Machine Learning techniques to time series data in order to deliver highly accurate forecasts. For instance, Uber critically relies on marketplace forecasts to direct employed drivers to areas of high demand in order to increase trip counts and revenues. This field has historically been taken up by traditional statistical forecasting methods. Machine learning methods have been developed and refined to the point that they now pose a serious challenge to purely statistical forecasting methods in the area of forecasting. Breiman et al. (2001) argue that there are two cultures with respect to statistical modeling of data which he labels as data modeling and algorithmic modeling. The first approach is based on the assumption of a stochastic data model and modeling response variables as a function of predictor variables, noise and parameters. On the other hand, the second approach is based on finding a function $f(\boldsymbol{x})$, an unknown algorithm, which operates on $\boldsymbol{x}$ to predict the response variables $\boldsymbol{y}$. The first modeling approach is the one underlying traditional statistical methods and the second approach, algorithmic modeling, underlies Machine Learning methods. Ahmed, Atiya, Gayar, and El-Shishiny (2010) come up with a large-scale comparison study based on the M3 competition dataset that provides an overview of commonly used machine learning models for time series forecasting. The most compelling evidence for the importance of machine learning methods in the context of forecasting is the fact that the best performing model emerging from the 2020 M4 forecasting competition, which involves 100,000 time series, was a combination of Exponential Smoothing (ETS) methods and a recurrent neural network (RNN) (Makridakis, Spiliotis, & Assimakopoulos, 2020). Januschowski et al. (2020) distinguish among methods to forecast a large number of similar series by estimating parameters jointly for all series (global methods) and methods which estimate parameters independently for each series (local methods). The aforementioned winning forecasting model provided by Smyl (2020) consists of a globally trained Exponential Smoothing model and locally trained Long Short

Term Memory neural networks (LSTMs). However, since we have collected two individual data sets for our analyses, we are concerned with local modeling.

For the purpose of this paper, the field of Smart City is chosen as a contemporary application domain of forecasting methods. The concept of a Smart City refers to the use of Information and Communication Technologies (ICT) in order to analyze and integrate the growing amount of available data on the ecosystems of our cities with the goal of making central infrastructure components more interconnected and efficient, which could significantly improve residents' quality of life in the cities of the future. New York City is a prime example of a city with good data availability and the commitment to develop into a Smart City. One major challenge which is faced by its Intelligent Transportation Systems (ITS) is the accurate prediction of traffic flows in order to mitigate congestion and the emission of pollutants.

Another forecasting challenge is faced by the public health sector, namely that of Emergency Medical Services (EMS) demand forecasting, which is concerned with predicting the number of emergency incidents that require the dispatch of ambulances and medical personnel. Accurate forecasts can help to ensure that the limited number of resources available can be used efficiently to deal with every incident.

Research design. Our research design with the aim to compare the forecasting performance of traditional statistical methods and that of Machine Learning methods is based on the research strategy of a case study which involves collecting two sources of evidence in the form of high-frequency data from two Smart City domains. We empirically investigate how well the predictions of the considered forecasting models generalize to previously unseen data and compute test error metrics. We also compare the quality of prediction intervals by computing commonly used evaluation metrics. We conduct time series analysis, a statistical methodology which inherently tracks observations repeatedly collected at a certain interval over time.

Main contribution. The purpose of this paper is to compare and contrast existing traditional models for time series forecasting with more recently emerged machine learning models such as ensemble models as well as recurrent neural networks. We concentrate our analyses on high-frequency hourly data from a Smart City context. A common characteristic of high-frequency data is the presence of multiple seasonal patterns and we explore the performance of forecasting models that can account for this complexity. In the M4 competition, by contrast, hourly data form only a minor part of the collection of time series amounting to less than 1 percent.[1]

[1] Makridakis et al. (2020) state that only 414 time series out of the 100,000 used in the M4 competition are of hourly frequency.

For these purposes, data sets from New York City Open Data[2], which are publicly available, are retrieved and the aforementioned methods are put to use in the selected Smart City domains of traffic forecasting and EMS demand forecasting.
Our results can be of use to smart city planners within the domains of traffic prediction and EMS resource planning since we provide accurate high-frequency point forecasts at an hourly frequency and we also quantify the associated uncertainties through the provision of prediction intervals.

Structure of the paper. Our paper proceeds as follows: Section 2 is concerned with outlining the concept of a Smart City and summarizing the previous research that has been done in the fields of traffic forecasting (Section 2.1) as well as forecasting demand for Emergency Medical Services (Section 2.2). Section 3 gives a technical background on the forecasting methods which are employed in this paper. Section 3.1 provides an introduction regarding recurring terminology and evaluation metrics in the context of time series forecasting. Section 3.2 presents the theoretical background of traditional statistical models such as the Seasonal Autoregressive Integrated Moving Average model (SARIMA) as well as Exponential Smoothing models and its extensions. Section 3.3 contains a theoretical background on the functional principles and pitfalls of ensemble methods in a time series context. In Section 3.4, functional principles of neural network architectures, which are suitable for time series forecasting, are explained. Section 4 contains a description of the data sets employed in this paper. Analyses, results and robustness checks are presented in Section 5 and discussed in **??**. **??** concludes.

[2]See https://opendata.cityofnewyork.us/.

Section 2

Smart City

The concept of a smart city is hard to define universally and a large variety of definitions can be found in the literature. However, Meijer and Bolívar (2016) identify three main streams of literature when it comes to the way of thinking about smart cities: a technological, a human resources and a governance focus. The first stream of literature considers technology as a defining characteristic of a smart city. Information and Communication Technologies (ICT) are employed in order to make central infrastructure components and services of a city, such as city administration, health care, transportation or public safety, more intelligent, interconnected and efficient. The other two characteristics identified contribute to a more holistic understanding of a smart city by adding the importance of collaboration among the citizens and the importance of their provision of inputs, e.g. through mobile devices, to the big picture.

Ismagilova, Hughes, Dwivedi, and Raman (2019) add that the ICTs used to implement the urban development vision of a smart city include smart hardware devices such as smartphones, smart vehicles and wireless sensors, but also comprise mobile networks and data storage technologies such as cloud platforms as well as software applications such as big data analytical tools.
Tascikaraoglu (2018) evaluates the role of forecasting methods in smart city applications and states that the importance of forecasting in this context stems from the necessity to obtain knowledge about plausible future conditions of the smart systems that are supposed to be managed efficiently, such as intelligent transportation systems or electric power systems. The author also argues that statistical and Machine Learning based modeling approaches to forecasting based on historical data are more adequate for the dynamic management of smart systems than mathematical approaches aimed at describing physical processes, which could also be considered when trying to forecast a related variable.

For our forecasting analyses we pick two relevant domains from the Smart City context which are traffic flow forecasting and forecasting the demand for Emergency Medical Services (EMS), for which we collect data from the NYC Open Data platform. For both of these data sets we will consider the following forecasting methods which are compared in Table 2.1. The comparison is structured by model capabilities such as handling nonstationarities, nonlinearities as well as multiple seasonal patterns.

Table 2.1: Comparison of prediction models considered in this paper

Model	Model type	Further regressors	Multiple seasonal patterns	Nonlinearities	Nonstationarity
SARIMA	parametric				✓
ETS/Bagged ETS	parametric			✓	✓
BATS/TBATS	parametric		✓	✓	✓
LGBM	non-parametric	✓	✓	✓	✓
FFNN	non-parametric	✓	✓	✓	✓
LSTM	non-parametric	✓	✓	✓	✓
GRU	non-parametric	✓	✓	✓	✓

Notes: This table contains a comparison of the prediction models considered in this paper and compares their respective features. Note that differencing or seasonal differencing might be required in order for the SARIMA model to be able to handle nonstationarity. The SARIMA model could be extended to include further regressors (SARIMAX model) or a spatial dimension (STARIMA model) but those two models are outside of the research space of this paper. The BATS and TBATS models can account for multiple seasonalities by estimating several seasonal parameters explicitly, whereas the neural networks account for them implicitly by learning the relationship through a complex network architecture. For explanations of the model acronyms, please refer to the List of Acronyms.

2.1 Traffic Forecasting

Understanding and forecasting future traffic conditions, for example as measured by traffic flows, travel times or traffic speeds, is a critical need of the transportation community and the quality of traffic information determines the success of Intelligent Transportation Systems. Stochastic characteristics of traffic flows make the forecasting task a challenging one but widely deployed traffic sensors ensure sufficient data availability and coverage.
Ma, Tao, Wang, Yu, and Wang (2015) argue that accurate traffic forecasts can be used for improving traffic safety, reducing congestion as well as rescheduling and preplanning routes for travelers. They also state that traffic forecasting methods have shifted away from traditional statistical models towards computational intelligence approaches such as neural networks because the latter are more capable of processing outliers, missing or noisy data and because they

require little or no assumptions about the input variables[1].

Nagy and Simon (2018) provide a comprehensive overview of commonly employed forecasting methods for traffic flow predictions in a Smart City context. They state that the traffic data model can range from simple scalar models based on data from a fixed position sensor to either time-space or region matrix models which identify both spatial and temporal correlations. Nagy and Simon (2018) categorize the traffic prediction models by the capabilities to handle nonstationarity, nonlinearities as well as the inclusion of a spatial dimension[2].
When considering the traditional statistical ARIMA model variants, we rule out the multivariate STARIMA model mentioned by Nagy and Simon (2018), since we limit our scope to univariate time series forecasting. The SARIMA model can be augmented to a SARIMAX model which can incorporate further regressors but we will limit our analysis to lagged values of the collected time series data in order to make the models better comparable since the ETS model does not allow for the inclusion of external regressors. Thus, we will use an automated SARIMA model as a first benchmark model but we will also manually refine an SARIMA model to be used as a candidate model from the traditional statistical modeling field. When considering the spectrum of neural network models, we will use the most suitable recurrent neural network architectures for time series forecasting, the LSTM and the GRU neural networks as candidate models from the Machine Learning field and take the non-recurrent Feedforward neural network (FFNN) as a benchmark model. The automated ETS model will also be considered as a benchmark model. To our knowledge, there have not been any extensive studies on traffic forecasting using ensemble models, so we will deploy the hybrid BaggedETS model as well as a Light gradient boosting machine (LGBM) model to investigate this gap in current research.

Related work. Vlahogianni, Karlaftis, and Golias (2014) summarize the studies on traffic forecasting which are not limited to a Smart City context for the period from 2003 to 2013 and emphasize the importance of choosing a suitable forecast horizon that matches the needs of traffic management systems. They also state that a higher data resolution is correlated with a higher amount of noise in the time series on traffic data which makes the development of traffic forecasting models more difficult. According to Fu, Zhang, and Li (2016), existing models such as ARIMA models cannot accurately describe the stochastic and non-linear nature of traffic flows. The authors therefore employ two deep-learning based models, the LSTM and the GRU, in order to forecast short-term traffic flows in the Bay Area of California in the U.S using sensor data. They find that the GRU outperforms both the ARIMA model and the LSTM neural network model.

[1]See Vlahogianni, Golias, and Karlaftis (2004, p. 542) for details on the assumptions on input variables and further characteristics of neural network, ARIMA and Smoothing models.

[2]See Nagy and Simon (2018, p. 157) for the full comparative list of traffic prediction models.

Zhao, Chen, Wu, Chen, and Liu (2017) employ an LSTM to forecast the volume of short-term traffic in Peking, i.e. they consider aggregated forecasting periods of 15 to 60 minutes, and compare the forecasting results to those of a regular RNN and of an Autoregressive Integrated Moving Average model (ARIMA) model. They find that the LSTM consistently outperforms the ARIMA model and that the regular RNN forecast error increases dramatically compared to the LSTM model as the forecasting period increases.

2.2 EMS forecasting

One key responsibility of governments of a specific local region or a city is to provide Emergency Medical Services (EMS) to the citizens in that specific area. EMS can be described as a system at the intersection of public safety, health care and public health which provides pre-hospital treatment of medical emergency incidents leading to serious injuries and transport to institutions with the capabilities to provide professional medical care. Aringhieri, Bruni, Khodaparasti, and van Essen (2017) state that research of EMS planning is centered around four main topics: demand forecasting, response times and workload, performance measurement as well as the determination of station locations, which requires the allocation of ambulances to stations based on changes in demand and travel times.
According to Aringhieri et al. (2017), forecasting demand for EMS is important in order to ensure that enough resources are available to meet emergency demand requests. These resources include ambulance cars, paramedics and medical doctors.
The data availability for EMS demand forecasting can generally considered to be good as EMS systems tend to be obliged to record data on the emergency calls and the dispatched ambulance cars. Commonly, recorded data include a spatial as well as a temporal dimension.

Related work. Several studies found in the EMS demand forecasting related literature opt to forecast emergency call volumes. For instance, Baker and Fitzpatrick (1986) employ an Exponential Smoothing Model to forecast EMS calls. Channouf, L'Ecuyer, Ingolfsson, and Avramidis (2007) also forecast daily and hourly EMS call volumes and employ a doubly seasonal ARIMA model. On the other hand, Setzler, Saydam, and Park (2009) produce forecasts for hourly and 3 hourly time intervals for specific areas of 4 by 4 square mile regions and employ a FFNN model. While ambulance fleet management can be facilitated by EMS demand forecasts, Setzler et al. (2009) argue that real time repositioning plans rely more on actual demand patterns than on demand forecasts.
In this paper, we will focus on forecasting hourly ambulance dispatch volumes as a proxy for EMS demand.

To the best of our knowledge, neither one of the most recent forecasting models such as the TBATS model from the statistical field or the LGBM boosting model as well as the LSTM and GRU recurrent neural networks form the Machine Learning field have been applied to this forecasting task. We therefore aim to close this gap in research and compare the quality of point forecasts and PI measures for the same spectrum of models that has been employed for the purposes of traffic flow forecasting. Note that forecasting volumes of emergency department visits is a closely related research topic but it is rather in the planning sphere of specific hospitals than in that of a smart city governance system.

Section 3

Forecasting Methods

3.1 Forecasting Terminology and Evaluation Metrics

Quantitative forecasting methods can be applied if numerical information about the past is available and if it is reasonable to assume that past patterns will continue into the future (R. J. Hyndman & Athanasopoulos, 2018). This paper is concerned with quantitative prediction problems related to time series data from a Smart City context. The model to be used in forecasting depends on the accuracy of the competing candidate models and on the availability of the data. Generally, our data of interest has to be split up into a training set, a validation set, which is used for fine tuning the hyperparameters of the forecasting model, and finally a test set, which is held out in order to be exclusively used for forecasting model evaluation. In supervised learning, Ng (2019) states that the goal is to learn an unknown function $f : \mathcal{X} \rightarrow \mathcal{Y}$ such that $f(\boldsymbol{x})$ is a good predictor for the corresponding value of $\boldsymbol{y}$. $\mathcal{X}$ and $\mathcal{Y}$ denote the feature space, i.e. the set of input values and the output space, respectively. This unknown function is also often referred to as hypothesis. An **explanatory model** for the training set can be set up as follows:

$$\boldsymbol{y} = f(\boldsymbol{X}) + \boldsymbol{\epsilon}, \tag{3.1}$$

where $\boldsymbol{y} \in \mathbb{R}^n$ is a vector of observed values of a quantitative prediction target variable such that $\boldsymbol{y} = (y^{(i)}; i = 1, \ldots, n)$. The superscript simply serves as an index into the training set. The design matrix $\boldsymbol{X} = (\boldsymbol{x_1 x_2} \ldots \boldsymbol{x_p})$ contains the set of p predictor vectors, which are called feature vectors in the field of Data Science, where $\boldsymbol{x}^{\boldsymbol{T}}_{\boldsymbol{j \in \{1,\ldots,p\}}} = (x_j^{(i)}; i = 1, \ldots, n)$ and $\boldsymbol{X} \in \mathbb{R}^{n \times p}$. $\boldsymbol{\epsilon} \in \mathbb{R}^n$ represents a vector of white noise error terms.
According to Ng (2019), the training set contains a list of n **training examples**, $\{(\boldsymbol{x}^{(i)}, y^{(i)}); i = 1, \ldots, n\}$.

Since the future prediction target is unknown, we can think of it as a random variable Y which could take on one in a range of possible future values. According to R. J. Hyndman and Athanasopoulos (2018), a **point forecast** is commonly understood as the "middle" of the range of possible future values as measured by the conditional mean. Moreover, it is also common practice to give a **prediction interval** to illustrate the range of future values that this random variable could take on with a specific probability. Chatfield (2001) defines a prediction interval as the interval between the upper and the lower limit of an interval estimate for an unknown future value, where the future value can be considered as a random variable at the point in time when the forecast is made.
Mathematically, we will denote the **forecast distribution**, which is the set of values that the random variable Y can take on and its corresponding probabilities of occurrence, as the forecast probability mass function of obtaining a discrete forecast of $\hat{y}_t$, which we denote by

$$p_Y(\hat{y}_t) = \mathbb{P}(Y = \hat{y}_t \mid \mathcal{I}), \tag{3.2}$$

where $\mathcal{I}$ denotes all past information available to forecast y_t. Chatfield (2001) states that it is also possible to find the complete probability distribution of a future value, which is called density forecasting, but this topic is not pursued further throughout this paper.
The h-step ahead forecast at time T is denoted by the conditional expectation

$$\hat{y}_{T+h} = \mathbb{E}[y_{T+h} \mid y_T, y_{T-1}, ..., y_1], \tag{3.3}$$

where h is the **forecast horizon** and T is the **forecast origin**.
In order to describe a forecasting model holistically, the estimation procedure for the model, the chosen loss function for model evaluation and the model selection procedure have to be explained.
To evaluate the usefulness of a forecast, a **loss function** has to be specified to measure how concerned we are about the fact that a forecast is off at a particular point in time (Hamilton, 1994). A very common choice is the **Mean squared error (MSE)** which is denoted as follows:

$$MSE(\hat{y}_{T+h|T}) = \mathbb{E}[(y_{T+h} - \hat{y}_{T+h|T})^2]. \tag{3.4}$$

The MSE is based on the **forecast error** which is defined as
$e_{T+h} = y_{T+h} - \hat{y}_{T+h|T}$, i.e. the difference between the observed value and its forecast. According to R. J. Hyndman and Athanasopoulos (2018), the difference between forecast errors and residuals of a regression model is that the former are computed on the test set instead of the training set. In this example, the test set is $\{y_{T+1}, y_{T+2}, ..., y_{T+h}\}$ and the training set is $\{y_1, ..., y_T\}$.

We refer to composite functions based on the forecast error as errors and those functions which are used as loss functions to measure forecasting model accuracy can be categorized into scale-dependent errors, percentage errors and scaled errors.
R. J. Hyndman and Athanasopoulos (2018) state that the first category includes errors that are on the same scale as the data, which means that those errors cannot be used for comparisons across time series with different units. The second category of percentage errors has the advantage of being unit-free, which means that forecasting performance across different data sets can be compared.

One of the most common scaled-dependent errors is the **Root mean squared error (RMSE)**, which is defined as

$$RMSE(\hat{y}_{T+h|T}) = \mathbb{E}[\sqrt{(y_{T+h} - \hat{y}_{T+h|T})^2}]. \tag{3.5}$$

Another popular scale-dependent error is the **Mean absolute error (MAE)**, which is computed as follows:

$$MAE(\hat{y}_{T+h|T}) = \mathbb{E}[\left|y_{T+h} - \hat{y}_{T+h|T}\right|]. \tag{3.6}$$

R. J. Hyndman and Koehler (2006) argue that the MSE and the RMSE measures are more sensitive to outliers than the MAE, which leads several researchers to prefer the latter as a a loss function to evaluate the accuracy of a forecasting method. The most frequently used percentage based error is the **Mean absoute percentage error (MAPE)**, which is defined as

$$MAPE(\hat{y}_{T+h|T}) = \mathbb{E}[\left|100\frac{y_{T+h} - \hat{y}_{T+h|T}}{y_{T+h}}\right|]. \tag{3.7}$$

R. J. Hyndman and Koehler (2006) compare accuracy measures for univariate time series and propose to use the **Mean absoute scaled error (MASE)** as a standard measure for a comparison of the forecast accuracy across multiple time series because they find that it performs well even when the data are close to zero or negative. The MASE belongs to the third category of scaled errors, all of which have the property that they indicate whether the forecast to be evaluated provides a better forecast than the average naive forecast based on the training data, which is the case if the value of the scaled error is less than 1. The MASE is defined as

$$MASE(\hat{y}_{T+h|T}) = \frac{\mathbb{E}[\left|y_{T+h} - \hat{y}_{T+h|T}\right|]}{\frac{1}{T-1}\sum_{t=2}^{T}|y_{t+h} - y_{t+h-1}|}, \tag{3.8}$$

where the numerator of the MASE is the MAE and the denominator is equivalent to a naive forecast. This shows how the MASE compares the MAE of the forecast to be evaluated to the

MAE of a naive forecast.
However, the denominator of the MASE, which contains the baseline forecasting model to which the MAE is compared, can be adapted to any model which is suitable for the forecasting task. If the data exhibits **seasonality**, which Jebb, Tay, Wang, and Huang (2015) define as any regular pattern of fluctuation in the level of the series associated with calendar effects, a seasonal naive model can be used as a baseline model. According to R. J. Hyndman and Athanasopoulos (2018), the **seasonal naive model** is given by

$$\hat{y}_{T+h|T} = y_{T+h-m(k+1)}, \tag{3.9}$$

where m is the **seasonal period**, which is also referred to as the seasonal frequency. k is the integer part of $(h-1)/m$, i.e. the number of complete seasonal periods prior to time $T+h$.

Cross validation is a common resampling method, i.e. a method of repeatedly drawing samples from a training set and then refitting a statistical learning method in order to obtain further information about the fitted model. According to James, Witten, Hastie, and Tibshirani (2013), cross-validation can be used for both model assessment and model selection. The former is achieved by estimating the test error of the statistical learning method for performance evaluation and the latter can be accomplished by tuning hyperparameters through a grid search cross-validation in such a way that a chosen error measure is minimized on the validation set. A k-fold cross validation involves the random partitioning of the entire data set into k folds of approximately equal size, where the first fold is used as a validation set and the model of interest is then fitted on the other $k-1$ folds. An error metric such as the MSE is then chosen as CV scoring function and then computed for the data in the current validation set. The process is subsequently repeated k times, where every fold or subset is used as a validation set once and the other $k-1$ subsets are used to fit the model. According to James et al. (2013), the **k-fold cross validation estimate** or **CV score**, which is the average of the k estimated validation errors, is given by

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i. \tag{3.10}$$

When time series data are considered, using a traditional k-fold Cross Validation (CV) is problematic because of temporal dependencies in the time series $\{y_t\}_{t=1}^{n}$. As Bergmeir and Benítez (2012) point out, when considering autoregression models applied to time series data, the assumption of cross-validation that the data are independently and identically distributed is violated in case autocorrelation is present in the data. Bergmeir, Hyndman, and Koo (2018) argue that a traditional k-fold CV is invalidated by removing k randomly chosen numbers from the series $\{y_t\}$ because of correlation between the errors in the training and test sets, which

would introduce a bias in the traditional CV procedure.

Generally, the time series is split into a training set and a test set and then the training set is further split into two subsets, a training subset and a validation set. The validation set is sometimes also referred to as the development set, since it is used to develop the best model, which can then subsequently be used on the test set for final evaluation. A graphical comparison between the regular k-fold cross validation method that is commonly deployed for cross-sectional data and the Time Series Split cross validation method both carried out on the training set is shown in Figure 3.1. The temporal order of the time series can be preserved by using the latter CV method since the training set is split into a training subset and a validation set at each iteration and the successive training subsets are supersets of the preceding training subsets.

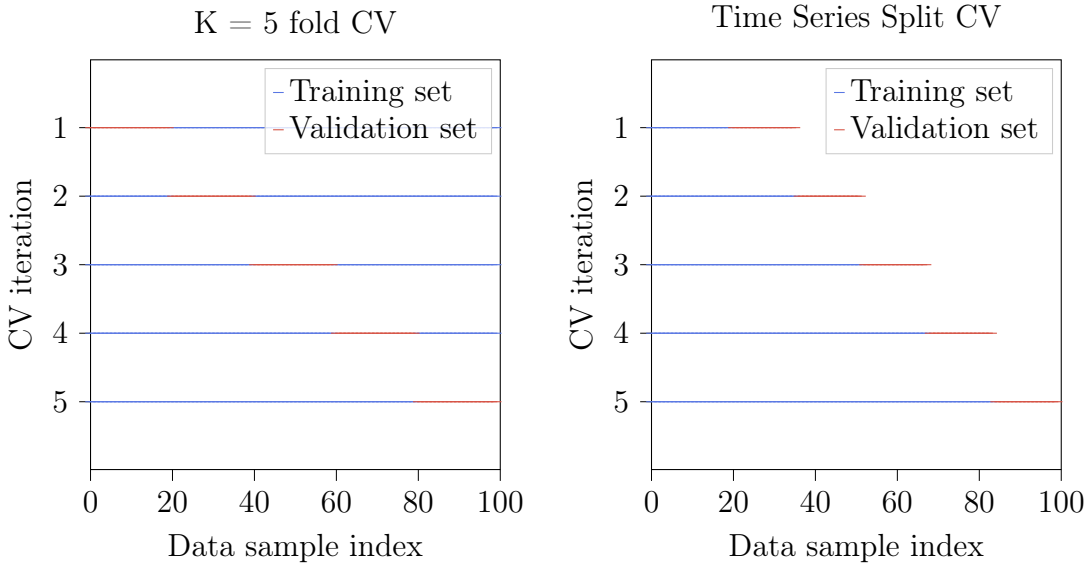

Figure 3.1: Comparison of K fold cross validation and Time Series Split cross validation

Notes: This figure compares the regular $K = 5$ fold cross validation shown in the left panel against the time series split cross validation, which is shown in the right panel. The time series split procedure involves the split of the training set into two subsets at each iteration, the training set and the validation set. However, the split is done under the condition that the validation set (shown in red) is always ahead of the training subset (shown in blue) in order to preserve the temporal order of the series. Note that the data cannot be randomly shuffled either, which is a difference to cross-sectional data. The Time Series Split cross validation method can be implemented in Python by means of the *TimeSeriesSplit* function of the *sklearn* library.

A further area of interest that requires evaluation metrics is the assessment of a Prediction interval (PI). Khosravi, Nahavandi, Creighton, and Atiya (2011) evaluate the four main methods found in the neural network literature to construct prediction intervals and present the

commonly used assessment measures to evaluate the quality of those prediction intervals.

A key aspect of prediction intervals is their coverage probability. The **Prediction Interval Coverage Probability (PICP)** measure constructs a probability that indicates how many of the target values y_i in the test set are covered by the constructed prediction intervals with i-th upper and lower bounds U_i and L_i. The PICP is given by

$$PICP = \frac{1}{n_{test}} \sum_{i=1}^{n_{test}} c_i, \tag{3.11}$$

where n_{test} is the number of samples in the test set and the coverage of the i-th prediction interval is given by

$$c_i = \begin{cases} 1, & y_i \in [L_i, U_i] \\ 0, & y_i \notin [L_i, U_i] \end{cases} . \tag{3.12}$$

Khosravi et al. (2011) state that the PICP should be close to or larger than the desired nominal confidence level of the prediction intervals to obtain reliable PIs and that an increase in the width of the PIs can increase the PICP but they also point out that choosing too wide PIs is not useful in practice since the variation of the targets cannot be seen anymore in that case. The width of prediction intervals is commonly assessed using the **Mean Prediction Interval Width (MPIW)**, which quantifies the average width of the prediction intervals as follows:

$$MPIW = \frac{1}{n_{test}} \sum_{i=1}^{n_{test}} (U_i - L_i). \tag{3.13}$$

Khosravi et al. (2011) also argue that the MPIW can be normalized by the range of the target values in order to be able to compare PIs across different data sets. The final assessment measure for PIs is the **coverage-width based criterion (CWC)**, which evaluates a PI based on the trade off between informativeness through narrow prediction intervals and correctness achieved through a high coverage probability. Khosravi et al. (2011) define the CWC as

$$CWC = NMPIW(1 + \gamma(PICP) * e^{-\xi(PICP-(1-\alpha))}), \tag{3.14}$$

where $NMPIW$ is the $MPIW$ normalized by the range of the target variable and $\gamma(PICP)$ is defined as

$$\gamma(PICP) = \begin{cases} 1, & PICP < (1-\alpha) \\ 0, & PICP \geq (1-\alpha) \end{cases} , \tag{3.15}$$

where α is the desired significance level. The term $\gamma(PICP)$ eliminates the exponential term

in Equation (3.14) when the PICP is greater than or equal to the nominal confidence level $1-\alpha$. The purpose of the exponential term is to penalize the violation of coverage probabilities, where ξ magnifies divergences between the PICP and the nominal confidence level $1-\alpha$. ξ and $(1-\alpha)$ are hyperparameters which are used to control the jump of the CWC. Note that a lower CWC value indicates a better PI quality as measured by both coverage probability and width of the prediction interval.

3.2 Traditional Statistical Models

This section provides a theoretical background for the traditional statistical models considered in this paper including the model equations and the necessary assumptions, the model selection procedure and the parameter estimation. We elaborate on the SARIMA model, the ETS model as well as its extensions, the BATS and the TBATS model.

3.2.1 SARIMA Model

In the domain of the statistical analysis of time series one of the most traditional classes of models is the Autoregressive Integrated Moving Average model. This model class was already proposed by Box, Jenkins, Reinsel, and Ljung (2015) in the first edition of their book in 1970. The authors derived the likelihood function and were thus able to estimate the parameters of the model through maximum likelihood. In addition, their work also contained the full end-to-end modeling procedure, which included specification, estimation, diagnostics and forecasting. Generally, ARIMA models are based on the idea that autocorrelation can be modeled through lagged linear relations which leads to the two main components, the autoregressive part and the moving average part.

An **autoregressive (AR) model** is one in which the current value of a time series is regressed on the previous values from that same time series. The direct use of previous values to describe future values is one of the most basic but powerful tools in time series analysis. The time gap between two values is called the **lag** and the p-th lag of a variable is therefore the realization of that variable p time steps ago. Using the notion of lags, we can write the general **AR(p) model**, where p indicates the order of the model, as follows:

$$y_t = \zeta_0 + \zeta_1 y_{t-1} + \zeta_2 y_{t-2} + ... + \zeta_p y_{t-p} + \epsilon_t, \tag{3.16}$$

where $\epsilon_t \sim N(0, \sigma^2)$ and ζ_0 is the level of the process or as it is more commonly referred to, the intercept. By introducing the **backshift operator** B as the operation defined by $By_t = y_{t-1}$

we can express the p-th lag of the series $\{y_t\}$ as $B^p y_t = y_{t-p}$. Using this operator we can also define the **autoregressive polynomial** as $\zeta(B) = 1 - \zeta_1 B - \zeta_2 B^2 - \cdots - \zeta_p B^p$. Equation (3.16) can thus be written more concisely as:

$$\zeta(B) y_t = \zeta_0 + \epsilon_t. \tag{3.17}$$

An important aspect of AR processes is the concept of stability. This can best be shown by looking at a simple AR(1) process, which can be written as

$$\begin{aligned} & y_t = \zeta_0 + \zeta_1 y_{t-1} + \epsilon_t \\ \iff \quad & y_t(1 - \zeta_1 B) = \zeta_0 + \epsilon_t \\ \iff \quad & y_t = (1 - \zeta_1 B)^{-1}(\zeta_0 + \epsilon_t) \\ & = \sum_{j=0}^{\infty} (\zeta_1 B)^j (\zeta_0 + \epsilon_t) = \sum_{j=0}^{\infty} \zeta_1^j (\zeta_0 + \epsilon_{t-j}). \end{aligned} \tag{3.18}$$

Note that the representation as an infinite geometric series can be reached by applying the definition of a geometric series to Equation (3.18) and, according to Box et al. (2015), provided that the stability condition for ζ_1 is met. Again, using the properties of an infinite geometric series, we know that the convergence criterion is that the modulus of the common ratio must be less than 1. The common ratio of the AR(1) process is ζ_1 and when the convergence criterion is met we say that the process is stable. The formulation in Equation (3.18) is only valid when $|\zeta_1| < 1$ and the AR process is stable. If $|\zeta_1| > 1$ we say that the process is exploding.

An alternative way to define the stability condition of an AR process is by inspection of the characteristic roots of the autoregressive polynomial. The root of $1 - \zeta_1 B = 0$ is $B = \zeta_1^{-1}$, which is then equivalent to the root lying outside the unit circle, since $|\zeta_1| < 1$ implies that $\left|\zeta_1^{-1}\right| > 1$.
According to Box et al. (2015), a succinct way to characterize the stability conditions for any AR(p) process is to state that the characteristic roots of the autoregressive polynomial must lie outside the unit circle.

A **moving average (MA) model** uses a linear combination of lags of the white noise error process rather than the previous observed values to form the series y_t. The general **MA model of order** q is denoted by

$$\begin{aligned} y_t &= \psi_0 + \epsilon_t + \psi_1 \epsilon_{t-1} + \psi_2 \epsilon_{t-2} + ... + \psi_q \epsilon_{t-q} \\ &= \psi_0 + \psi(B)\epsilon_t \end{aligned}, \tag{3.19}$$

where ψ_0 is the intercept and $\psi(B) = 1 + \psi_1 B + \psi_2 B^2 + ... + \psi_q B^q$ is the **moving aver-**

age polynomial, which is defined in a similar manner as the autoregressive polynomial and $\epsilon_t \sim N(0, \sigma^2)$. Each value of y_t can thus be viewed as a weighted average of past errors.
It is possible to write any AR(p) model as a MA(∞) model. The reverse is also true if the MA(q) process is said to be invertible. The conditions that must be satisfied to ensure invertibility are similar to the stability conditions for an AR(p) process. According to Box et al. (2015), the invertibility condition for a MA(q) process is that the characteristic roots of the moving-average polynomial must lie outside the unit circle.

Despite the fact that both AR and MA processes can be transformed into the other under certain invertilibilty conditions, in practice and to achieve parsimony, it is often beneficial to combine the two models into a mixed model. This resulting model is called the **Autoregressive Moving Average Model (ARMA)**. Using the autoregressive polynomial and the moving average polynomial and following the notation of Shumway and Stoffer (2017), the ARMA(p,q) model of autoregressive order p and moving average order q can be concisely written as

$$\zeta(B)y_t = c + \psi(B)\epsilon_t, \tag{3.20}$$

where $c = (1 - \zeta_1 - \zeta_2 - \cdots - \zeta_p)\mu$, μ is the non-zero mean of the series and $\epsilon_t \sim N(0, \sigma^2)$. Note that the model given in Equation (3.20) is only valid under the assumption that the series $\{y_t\}$ is stationary or at least weakly stationary. According to Enders (2014), a time series is considered **weakly stationary** if its mean and all autocovariances are finite and independent of the time interval in which they are observed. Any finite MA process is thus stationary. However, that is not the case for AR processes. The stationarity conditions for an AR process are closely related to stability. In fact a sufficient condition for stationarity is that the process is stable. Therefore, Enders (2014) states that if all the characteristic roots of the autoregressive polynomial are outside the unit circle, then the process is both stable and stationary. Equation (3.20) will thus be stationary given that $\zeta(B) = 0$ has all its roots outside the unit circle.
If the time series $\{y_t\}$ is not stationary, a transformation is required to make it stationary before an ARMA(p,q) model can be applied. A common transformation is to take the differences between consecutive observations. This procedure is known as **differencing** and it is based on the utilization of the **backward difference operator**, which is defined as $\Delta y_t = y_t - y_{t-1}$. The d-th difference is then given by $\Delta^d y_t = (1 - B)^d y_t$ where d is the order of differencing. Enders (2014) states that differencing has the effect of stabilizing the mean and in addition, the d-th difference of a process with d unit roots is stationary. The generalization of ARMA(p,q) processes to nonstationary time series by including the procedure of differencing leads to the **autoregressive integrated moving average (ARIMA) model**. An general ARIMA model with autoregressive order p, an order of differencing of d and a moving average order of q is

denoted as ARIMA(p,d,q) and can be written as

$$\zeta(B)\Delta^d y_t = c + \psi(B)\epsilon_t, \tag{3.21}$$

where $\zeta(B)$ is the autoregressive polynomial, Δ^d is the d-th difference and $\psi(B)$ is the moving average polynomial. Equation (3.21) assumes that the d-th difference of y_t can be modeled by a stationary ARMA process of order (p, q).

Moreover, the ARIMA model can be modified further in order to take not only nonstationarity but also seasonality into account. If dependence on past trends tends to be most pronounced in a cyclic manner at some seasonal lag s, we can set up a multiplicative **seasonal integrated moving average (SARIMA) model** in the following way, adhering to the notation of Shumway and Stoffer (2017):

$$Z_P(B^s)\zeta(B)\Delta_s^D\Delta^d y_t = c + \Psi_Q(B^s)\psi(B)\epsilon_t. \tag{3.22}$$

This general SARIMA model is denoted as ARIMA$(p, d, q) \times (P, D, Q)_s$. The regular autoregressive polynomial is $\zeta(B)$ and the moving average polynomial is $\psi(B)$. In this model, we now also have seasonal autoregressive and moving average components denoted by $Z_P(B^s)$ and $\Psi_Q(B^s)$, respectively. The seasonal AR order is P and the seasonal MA order is Q. Finally, $\Delta_s^D = (1 - B^s)^D$ denotes the D-th seasonal difference and $\Delta^d = (1 - B)^d$ is the regular d-th difference.

The estimation of ARIMA models, which can be represented by an ARMA(p,q) model after an appropriate number of differencing operations has been carried out, is commonly done with maximum likelihood. According to Shumway and Stoffer (2017), we assume that we have n observations $y_1, \ldots, y_n$ from an invertible ARMA(p,q) process with initially known order parameters p and q. The goal is then to estimate the parameters contained in the model parameter vector

$$\boldsymbol{\theta}^T = (\mu, \zeta_1, \ldots, \zeta_p, \psi_1, \ldots, \psi_q), \tag{3.23}$$

where $\boldsymbol{\theta} \in \mathbb{R}^{p+q+1}$. μ is the non-zero mean of the series, ζ_p is the p-th AR parameter and ψ_q is the q-th MA parameter of the time series.
Shumway and Stoffer (2017) argue that the approach to write the likelihood in terms of the one-step ahead prediction errors $e_{t+1} = y_{t+1} - \hat{y}_{t+1|t}$ instead of as an explicit function of the model parameters is preferable. The likelihood function can be written as

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\theta}, \sigma_\epsilon^2) = \prod_{t=1}^{n} f(y_{t+1}|y_t, \ldots, y_1), \tag{3.24}$$

where the conditional distribution of y_{t+1} given $y_t, \ldots, y_1$ is assumed to be Gaussian with mean $y_{t+1|t}$ and variance $P_{t+1|t}$. ρ denotes the autocorrelation function and σ_ϵ^2 is the variance of the error term. Using the ARMA autocovariance, which is given by $\gamma(0) = \sigma_\epsilon^2 \sum_{j=0}^{\infty} \rho_j^2$, we can write the variance of the conditional distribution of the series as

$$P_{t+1|t} = \gamma(0) \prod_{j=1}^{t}(1 - \zeta_{jj}^2) = \sigma_\epsilon^2 * \underbrace{[\sum_{j=0}^{\infty} \rho_j^2][\prod_{j=1}^{t}(1 - \zeta_{jj}^2)]}_{r_t}, \tag{3.25}$$

where ζ_{jj} is the partial autocorrelation coefficient at lag j. Note that we can now express the terms of the variance of the Gaussian conditional distribution of y_t which are not dependent on ϵ_t as $r_1, \ldots, r_n$. The latter terms are computed recursively using $r_{t+1} = (1 - \zeta_{tt}^2) r_t$ with the initial condition $r_1 = \sum_{j=0}^{\infty} \rho_j^2$.
The **likelihood of the data** is then given by

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\theta}, \sigma_\epsilon^2) = (2\pi\sigma_\epsilon^2)^{-n/2} [r_1\boldsymbol{\theta} r_2\boldsymbol{\theta} \cdots r_n\boldsymbol{\theta}]^{-1/2} exp[-\frac{S(\boldsymbol{\theta})}{2\sigma_\epsilon^2}], \tag{3.26}$$

and can be maximized with respect to $\boldsymbol{\theta}$ and σ_ϵ^2 to obtain the maximum likelihood estimates of the ARMA model parameters. By maximizing the likelihood function, the goal of making our data of interest the most probable under our pre-specified distributional assumption, which is the Gaussian probability density in this particular case, can be accomplished. Note that the sum of the squared one-step ahead prediction errors, which is a function of the model parameters, is given by

$$S(\boldsymbol{\theta}) = \sum_{t=1}^{n} \frac{(y_{t+1} - \hat{y}_{t+1|t}(\boldsymbol{\theta}))^2}{r_t(\boldsymbol{\theta})}. \tag{3.27}$$

This maximum likelihood estimation is then commonly implemented numerically by means of the Newton-Rapson or the Scoring algorithm.

The estimation of the parameters contained in $\boldsymbol{\theta}$ is only part of the model selection process. The parameters which control the order of the autoregressive processes, the moving-average processes and the order of both the first and the seasonal differences are so called hyperparameters. These are chosen outside the estimation procedure based on either graphical heuristics, in-sample measures or out of sample measures. When Box et al. (2015) first introduced an end to end modeling approach, the **sample autocorrelation function** (ACF) and the **sample partial autocorrelation function** (PACF) were the principle tools. The ACF plots the auto-

correlation of the series by lag and the PACF plots the resulting autocorrelation at a specific lag after removing the effects of autocorrelations at shorter lags. Other more modern approaches aim to automate the selection of the hyperparameters. The *auto.arima()* function in *R* first selects the order of differencing based on a statistical test for the presence of a unit root such as the **Augmented Dickey Fuller test (ADF)** and thereafter selects the order of the remaining hyperparameters $\{p, q, P, Q\}$ based on a in-sample measure such as the AIC (R. J. Hyndman & Athanasopoulos, 2018) and is based on the Hyndman-Khandakar algorithm (R. J. Hyndman & Khandakar, 2008). The algorithm can be varied by specifying options for the *auto.arima* function such as the *stepwise* and the *approximation* options. Instead of traversing the model space in a stepwise manner and instead of approximating the information criteria, the algorithm will consider all possible model parameter combinations and refrain from approximations. However, the *auto.arima* function still leaves the search for outliers, transformations, as well as a visual inspection of the model residuals and a formal test for autocorrelation among the residuals to the end user of the algorithm (Petropoulos, Hyndman, & Bergmeir, 2018). The AIC criterion is based on the likelihood of the data, i.e. it is based on the same Gaussian likelihood that is stated in Equation (3.26), and it is given by

$$AIC = -2\,log(\mathcal{L}) + 2(p + q + k + 1), \tag{3.28}$$

where $(p + q + k + 1)$ is the number of parameters in the model including the variance of the residuals. Note that the order of regular differencing d cannot be determined by the AIC since the data on which the likelihood function is computed is changed by the differencing procedure and the AIC values of models of different orders of differencing are not comparable anymore.

Forecasting future values of the series $\{y_t\}$ using a SARIMA model is done by means of computing the expectation conditional on the information available up to the forecast origin T. A **one-step ahead forecast** can thus be be written as:

$$\mathbb{E}[y_{T+1} \mid y_T, y_{T-1}, \ldots, y_1] = \hat{y}_{T+1|T}. \tag{3.29}$$

As an illustrative example, let us consider the ARIMA$(2, 1, 1) \times (0, 0, 1)_{12}$ model, which we can expressed as

$$\begin{aligned} & \zeta(B)\Delta y_t = \Psi(B^S)\psi(B)\epsilon_t \\ \iff \quad & (1-\zeta_1 B-\zeta_2 B^2)(y_{t-1}-y_t) = (1+\Psi_1 B^{24})(1+\psi_1 B)\epsilon_t \\ \iff \quad & y_t = (1+\zeta_1)y_{t-1} - (\zeta_1-\zeta_2)y_{t-2} - \zeta_3 y_{t-3} + \epsilon_t + \psi_1\epsilon_{t-1} \\ & \qquad + \Psi_1\epsilon_{t-12} + \psi_1\Psi_1\epsilon_{t-13}. \end{aligned} \tag{3.30}$$

The conditional expectation of Equation (3.30), which constitutes a one-step ahead forecast using the aforementioned SARIMA model, is then given by

$$\hat{y}_{T+1|T} = (1+\hat{\zeta}_1)y_T - (\hat{\zeta}_1-\hat{\zeta}_2)y_{T-1} - \hat{\zeta}_3 y_{T-2} + \hat{\psi}_1\epsilon_T + \hat{\Psi}_1\epsilon_{T-11} + \hat{\psi}_1\hat{\Psi}_1\epsilon_{T-12}. \tag{3.31}$$

Equation (3.31) can be considered the **forecast equation** from which, according to R. J. Hyndman and Athanasopoulos (2018), future point forecasts can be obtained using the estimated model parameters from the fitted model, forecasts for future observations, zero for future errors and the respective residual terms for past errors. If multi-step ahead forecasts are required, the same procedure can be used. For example, the conditional expectation of y_{T+2} is computed by replacing y_{T+1} by $\hat{y}_{T+1|T}$. Forecasts of length h can be obtained in a recursive manner using the previous forecasts instead of the unobserved true values as shown in the example. For any stationary SARIMA model, the conditional forecast of y_{T+h} converges to the unconditional mean as $h \to \infty$ (Enders, 2014).

Even though point forecasts produced by the SARIMA model can be considered unbiased if the conditional expectation of the error terms is zero, they tend to be inaccurate. Therefore, the properties of uncertainty surrounding the point forecast and the forecast errors are important. According to Enders (2014), the variance of the forecast errors is an increasing function of h and the forecast error variance converges to the unconditional variance of the $\{y_t\}$ sequence. R. J. Hyndman and Athanasopoulos (2018) argue that prediction intervals of ARIMA models are based on the assumptions that the estimated model's residuals ϵ_t are uncorrelated and normally distributed and that the prediction intervals may not be correct if either one of these two assumptions does not hold. They also state that ARIMA PIs will increase as the forecast horizon increases and that these prediction intervals are too narrow since only the variation in the errors has been accounted for but not the variation in the model parameter estimates and the model order. A 95% **multi-step prediction interval** is given by

$$\hat{y}_{T+h|T} \pm 1.96\sqrt{\hat{\sigma}_h^2}, \tag{3.32}$$

where $\hat{\sigma}_h^2$ is the estimated forecast variance for an h-step ahead forecast. Note that the exact shape of $\hat{y}_{T+h|T}$ and $\hat{\sigma}_h^2$ depends on the specification of the SARIMA model and the forecast horizon h. According to R. J. Hyndman and Athanasopoulos (2018), in the simplest case, when we consider an ARIMA(0,0,q) model, the estimated forecast variance $\hat{\sigma}_h^2 = \hat{\sigma}_\epsilon^2[1 + \sum_{i=1}^{h-1} \hat{\psi}_i^2]$, where $\hat{\sigma}_\epsilon^2$ is the variance of the estimated model's residuals.[1]

3.2.2 Exponential Smoothing Model

Exponential Smoothing Models (ETS) have first been proposed in the 1950s (Brown, 1959). However, a comprehensive modeling framework including stochastic models, prediction intervals and procedures for model selection have not been developed until 2002. R. J. Hyndman, Koehler, Snyder, and Grose (2002) provide an automatic forecasting framework for selecting the best model out of the taxonomy of 30 state space models which minimizes the AIC information criterion for a given time series. R. J. Hyndman, Koehler, Ord, and Snyder (2008) argue that one benefit of state space models is that they are easy to use in a fully automated way.

State space models underlie exponential smoothing models and for every smoothing method there are two models, one with additive errors and one with multiplicative errors. Therefore, the common notation for ETS methods is a triplet (E, T, S), the components of which represent error type, trend type and seasonality type, respectively. In total, there are 30 ETS models when considering the two possible error types for every single one of the 15 exponential smoothing methods shown in Table 3.1, which are classified by trend and seasonality. The 30 ETS model combinations arise from the following modeling choices: The error term can be modeled as additive or multiplicative $\{A, M\}$. The set of trend modeling choices is given by $\{N, A, A_d, M, M_d\}$ and seasonality can be set to none or it can be modeled as additive or multiplicative, which gives the set $\{N, A, M\}$.

The trend component of an ETS model is composed of a level term l_t and a growth term b_t which can include a dampening parameter ϕ such that $0 \leq \phi \leq 1$. The dampening parameter ϕ can be used to dampen the trend as the length of the forecast horizon increases. Thus, combining the trend and growth patterns yields five different forecast trend types T_h over h time periods:

$$T_h = l \tag{3.33}$$

$$T_h = l + bh \tag{3.34}$$

[1] See Brockwell and Davis (2016) for prediction intervals of more complicated models.

$$T_h = l + (\phi + \phi^2 + \ldots + \phi^h)b \tag{3.35}$$

$$T_h = lb^h \tag{3.36}$$

$$T_h = lb^{(\phi+\phi^2+\ldots+\phi^h)}. \tag{3.37}$$

Table 3.1 is based on the trend type equations given by Equations (3.33) to (3.37), which denote the trend types "None", "Additive", "Additive damped", "Multiplicative" and "Multiplicative damped".

The main idea behind exponential smoothing methods is to provide an extrapolation procedure that uses a weighted average of past observations to forecast future values of a time series in which the weights are decreased exponentially as the observations become older in order to give more weight to more recent observations (Armstrong, 2001). The challenge with ETS models is to select the right method to approximate the data generating process accurately by means of specifying the level, trend and seasonality components of the time series.
If the data don't exhibit a trend or a seasonal pattern, simple exponential smoothing is a an appropriate choice. If a linear trend is present, then Holt's model for a linear or damped trend is appropriate. However, if there are also seasonality effects present in the data, then the **Holt-Winters' method** is the right choice. The model consists of the following three smoothing equations which model the respective components of the series:

$$l_t = \alpha \frac{y_t}{s_{t-m}} + (1-\alpha)(l_{t-1} + b_{t-1}) \tag{3.38}$$

$$b_t = \beta^*(l_t - l_{t-1}) + (1-\beta^*)b_{t-1} \tag{3.39}$$

$$s_t = \gamma y_t/(l_{t-1} + b_{t-1}) + (1-\gamma)s_{t-m}, \tag{3.40}$$

where s_t denotes the seasonal term. Finally, the forecast equation for the Holt-Winters' method with multiplicative seasonality is

$$\hat{y}_{T+h|T} = (l_T + b_T h) * s_{T-m+h_m^+}, \tag{3.41}$$

where h_m^+ is the number of remaining times in the forecast period up to and including time h, the forecast horizon. m is the number of periods in each season.

Table 3.1: Classification of Exponential Smoothing Methods by trend and seasonality type

Trend Component	**Seasonal Component**		
	None (N)	*Additive (A)*	*Multiplicative (M)*
None (N)	(N,N)	(N,A)	(N,M)
Additive (A)	(A,N)	(A,A)	(A,M)
Additive damped (A_d)	(A_d,N)	(A_d,A)	(A_d,M)
Multiplicative (M)	(M,N)	(M,A)	(M,M)
Multiplicative damped (M_d)	(M_d,N)	(M_d,A)	(M_d,M)

Notes: This table contains the classification taxonomy of Exponential Smoothing methods by trend and seasonality type used by R. J. Hyndman et al. (2008). Note that the (N, N) model is equivalent to the Simple Exponential Smoothing model. Holt's method for linear trends is represented by the (A, N) model and the (A_d, N) model represents Holt's method for damped trends. Finally, the Holt-Winters' method for additive seasonality is given by the (A, A) model and the Holt-Winters' method for multiplicative seasonality is given by the (A, M) model.

R. J. Hyndman et al. (2008) also argues that if the same parameter values are used for one of the methods from the classification and only the error type is varied from additive to multiplicative, the same point forecasts are produced but the prediction intervals will differ. Note that an exponential smoothing method is an algorithm that produces a point forecast only, whereas the underlying stochastic state space model will produce the same point forecast but can also be used to compute prediction intervals.

The non-linear innovations **state space model equations** proposed by R. J. Hyndman et al. (2008) on which all 30 Exponential smoothing methods are based are as follows:

$$y_t = \omega(\boldsymbol{x}_{t-1}) + r(\boldsymbol{x}_{t-1})\epsilon_t, \tag{3.42}$$

$$\boldsymbol{x}_t = \boldsymbol{f}(\boldsymbol{x}_{t-1}) + \boldsymbol{g}(\boldsymbol{x}_{t-1})\epsilon_t, \tag{3.43}$$

where $\{\epsilon_t\}$ is a white noise series of innovations. R. J. Hyndman et al. (2008) argues that we can assume this series to be distributed as
$\{\epsilon_t\} \sim \mathcal{N}(\mu_t = \omega(\boldsymbol{x}_{t-1}), \sigma^2)$ for convenient inferences if the structure of the data generating process does not conflict with this assumption, which would be the case if the series contained only non-negative values. $\boldsymbol{f}(\cdot)$ and $\boldsymbol{g}(\cdot)$ are vector-valued functions and $\omega(\cdot)$ and $r(\cdot)$ are scalar

functions with time-varying arguments.
y_t denotes an observation at time t and $\boldsymbol{x}_t$ denotes a state vector which is made up of unobserved components that describe the level, trend and seasonality of the time series.
The **state vector** is

$$\boldsymbol{x}_t^T = (l_t, b_t, s_{t-1}, \ldots, s_{t-m+1}), \tag{3.44}$$

where l_t is the level of the series, b_t is the slope of the series at time t and s_t is the seasonality component of the series at time t with m denoting the number of seasons per year.
R. J. Hyndman et al. (2008) state that Equation (3.42) describes the relationship between the unobserved states $\boldsymbol{x}_{t-1}$ and the observation y_t and is called the **observation equation**. Equation (3.43) models the evolution of the states, i.e. the unobserved components of the series, over time and is therefore called the **state equation**. The white noise innovation series is identical in the two equations, which is why the two equations can be considered a state space innovation model.
Note that the error component of the state space innovation model is parametrized by $r(\boldsymbol{x}_{t-1})$ in Equation (3.42). In case of additive errors $r(\boldsymbol{x}_{t-1}) = 1$, which means that $y_t = \mu_t + \epsilon_t$ and in case of multiplicative errors $r(\boldsymbol{x}_{t-1}) = \mu_t$, i.e. $y_t = \mu_t(1 + \epsilon_t)$.

As a simple non-linear example to illustrate the state space model equations we consider an ETS(M,N,N) model, i.e. a local level model with multiplicative errors and neither a trend nor a seasonality component. In this case, the components of the observation equation are $\omega(\boldsymbol{x}_{t-1}) = r(\boldsymbol{x}_{t-1}) = l_{t-1}$ and the components of the state equation are $\boldsymbol{f}(\boldsymbol{x}_{t-1}) = l_{t-1}$ and $\boldsymbol{g}(\boldsymbol{x}_{t-1}) = \alpha l_{t-1}$. Thus, we can derive the following two state space model equations:

$$y_t = \mu_t(1 + \epsilon_t) = l_{t-1}(1 + \epsilon_t) \tag{3.45}$$

$$l_t = l_{t-1}(1 + \alpha\epsilon_t), \tag{3.46}$$

which can be found in **??**. We can also derive the recursive formula for a point forecast given in **??** by eliminating the error term ϵ_t and we obtain

$$l_t = \alpha y_t + (1 - \alpha) l_{t-1}. \tag{3.47}$$

One-step-ahead point forecasts from ETS models are computed using the conditional expectation as follows:

$$\mathbb{E}[y_{T+1}|y_T, y_{T-1}, \ldots, y_1, \boldsymbol{x}_0] = \mathbb{E}[y_{T+1}|\boldsymbol{x}_T] = \hat{y}_{T+1|T} = \omega(\boldsymbol{x}_T). \tag{3.48}$$

In order to obtain multi-step ahead forecasts in a recursive way, a prediction is computed by Equation (3.48), the output of which is then used as an input to predict the subsequent time step and the innovations are updated by

$$\epsilon_{T+1} = (y_{T+1} - \hat{y}_{T+1|T})/r(\boldsymbol{x}_T). \tag{3.49}$$

Moreover, the state equation, Equation (3.43), is updated using the new innovation from Equation (3.49).
All formulae for the calculation of point forecasts through exponential smoothing methods are given in **??**.

Since point forecasts are simply predictions of future values expressed as a single number which give no indication of the associated uncertainty, we need to compute prediction intervals. R. J. Hyndman et al. (2008) argue that prediction intervals for linear state space models with homoskedastic errors that are assumed to be Gaussian can be computed from the forecast means, $\mu_{T+h|T}$, and the forecast variances $\nu_{T+h|T}$ in the following way:

$$\mu_{T+h|T} \pm z_{\alpha/2}\sqrt{\nu_{T+h|T}}, \tag{3.50}$$

where z_q is the q-th percentile of the Standard Normal distribution. However, note that the exact algebraic expressions for the forecast variances change if linear state space models with heteroskedastic errors are considered. Thus, the prediction intervals and the forecast distributions will change as well. If nonlinear seasonal state space models are considered, the exact values for the forecast means and variances are hard to compute if $h \geq m$, i.e. if the forecast horizon is greater than the number of periods in a season. Thus, approximate forecast means and forecast variances have to be used. R. J. Hyndman et al. (2008) argue that these approximations produce reasonably accurate results.
On the other hand, if the assumption of Gaussian distributed innovations is inadequate, a second approach for the computation of prediction intervals has to be used. This second approach is to simulate the forecast distribution and and prediction intervals by means of simulating future sample paths from the model, i.e. from Equation (3.42) and Equation (3.43) conditional on the final state $\boldsymbol{x}_T$. This is done by recursively generating the set of observations $\{y_t^{(i)}\}$ for $t = T+1, \ldots, T+h$ starting with $\boldsymbol{x}_T$. Each value for ϵ_t is taken from a random number generator which draws randomly generated values from an appropriate prespecified distribution, which does not have to be the Gaussian distribution. We repeat this procedure for $i = 1, \ldots, M$, where M is a large integer value that denotes the number of simulated sample paths. R. J. Hyndman et al. (2008) state that 5000 is a common choice for M and they also argue that the prediction distribution of $y_{T+h|T}$ can then be estimated from the vector of simulated values at

forecast horizon h, which is

$$\boldsymbol{y}_{T+h|T}^{T} = \{y_{T+h}^{(1)}, \ldots, y_{T+h}^{(M)}\}, \tag{3.51}$$

where $\boldsymbol{y}_{T+h|T} \in \mathbb{N}^{M\times 1}$. Finally, prediction intervals for these nonlinear state space models are then obtained from the quantiles of the of the simulated sample paths. The $100(1-\alpha)\%$ prediction interval for an h-step ahead forecast is then given by the $\alpha/2$ and the $1-\alpha/2$ quantiles of the vector of simulated values $\boldsymbol{y}_{T+h|T}$. R. J. Hyndman et al. (2008) argue that this non-analytical method is the only way of obtaining prediction intervals for those nonlinear subclasses of ETS models because no distributional assumptions are required when resampling the innovations ϵ_t. They also state that an advantage of simulated prediction intervals is that the uncertainty associated with estimating the model parameters can be taken into account since the sample paths are generated from the same model but with randomly varying parameters $\alpha, \beta, \gamma, \phi$ and $\boldsymbol{x}_0$.

The state space model equations that are used for the calculation of prediction intervals are given in **??** and **??**.

R. J. Hyndman et al. (2008) argue that certain combinations of error, trend and seasonality can lead to numerical difficulties and he therefore states that the multiplicative error models can be numerically unstable when the time series contains data with zeros or negative values.

The estimation of state space innovation models requires the estimation of the unknown initial states vector $\boldsymbol{x}_0$ and the estimation of the unknown model parameters α, β, γ and ϕ, which refer to the the three smoothing parameters and the dampening parameter, respectively, after the type of ETS model to be deployed has been selected.
The **initial states vector**, $\boldsymbol{x}_0 \in \mathbb{R}^{k\times 1}$, is defined as

$$\boldsymbol{x}_0^T = (l_0, b_0, s_{-1}, \ldots, s_{-m+1}) \tag{3.52}$$

and can be specified using ad-hoc values or heuristic schemes.[2]
These initial states are then refined by estimating them in conjunction with the model parameters.
The ETS model selection can be done by means of the forecast accuracy error measures introduced in Section 3.1, which are computed on the test set, but a penalized method such as the AIC, which is computed on the training set, is the better choice for selecting between the additive and the multiplicative error models since the AIC is based on the likelihood instead of

[2]For details on the latter see R. J. Hyndman et al. (2008, p. 23)

the one-step ahead forecasts. R. J. Hyndman et al. (2008) claim that the error measures cannot be used to select between additive and multiplicative error models of exponential smoothing methods since the two error types produce the same point forecasts. The AIC is given by

$$AIC = \mathcal{L}^*(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x_0}}) + 2q, \tag{3.53}$$

where q is the number of parameters in $\boldsymbol{\theta}$ plus the number of free states in $\boldsymbol{x_0}$.
The parameter vector of the likelihood is given by

$$\boldsymbol{\theta}^T = (\alpha, \beta, \gamma, \phi) \tag{3.54}$$

and we restrict the parameter space as follows:

$$0 \leq \alpha \leq 1, \quad 0 \leq \beta \leq \alpha, \quad 0 \leq \gamma \leq 1 - \alpha, \quad \text{and} \quad 0 \leq \phi \leq 1. \tag{3.55}$$

α controls the weights used on the current and past observations.
The likelihood also depends on the joint density of the series vector $\boldsymbol{y}^T = (y_1, \ldots, y_n)$, which is

$$p(\boldsymbol{y}|\boldsymbol{\theta}, \boldsymbol{x}_0, \sigma^2) = \prod_{t=1}^{n} p(\epsilon_t) / \left| r(\boldsymbol{x}_{t-1}) \right|, \tag{3.56}$$

where σ^2 is the variance of the innovations. Thus we have a weighted product of the densities of the individual innovations, $p(\epsilon_t)$.
If we use the assumption of Gaussian innovations, we can write the Gaussian likelihood function as

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{x_0}, \sigma^2|\boldsymbol{y}) = (2\pi\sigma^2)^{-n/2} \left| \prod_{t=1}^{n} r(\boldsymbol{x}_{t-1}) \right|^{-1} \exp\left(-\frac{1}{2} \sum_{t=1}^{n} \epsilon_t^2 / \sigma^2\right). \tag{3.57}$$

Note that by eliminating σ^2 from the Gaussian likelihood function, taking the log and multiplying by -2 we can proceed to derive the following likelihood function of the state space innovations model, which we want to minimize with respect to the main model parameter vector $\boldsymbol{\theta}$ and the initial states vector $\boldsymbol{x}_0$ in order to obtain the maximum likelihood estimates :

$$\mathcal{L}^*(\boldsymbol{\theta}, \boldsymbol{x_0}) = n \log\left(\sum_{t=1}^{n} \epsilon_t^2\right) + 2 \sum_{t=1}^{n} \log\left| r(\boldsymbol{x}_{t-1}) \right|. \tag{3.58}$$

Note that this likelihood function can accommodate heteroskedastic innovations and in case of homoskedastic innovations we simply have $r(\boldsymbol{x}_{t-1}) = 1$. R. J. Hyndman et al. (2008) also state that a poor choice of initial or seed values can lead to sub-optimal estimates and to higher computational loads.
Furthermore, stationarity of the data is not a required assumption for the usage of ETS models.

R. J. Hyndman et al. (2008) state that stationarity is in fact a rare property in exponential smoothing state space models.

3.2.3 BATS and TBATS Models

The state space models which underlie the ETS models described in Section 3.2.2 can also be extended further in order to enable ETS models to handle more complex seasonal patterns. Livera, Hyndman, and Snyder (2011) propose an **Exponential smoothing state space model with trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend and Seasonal components (TBATS)**, which can handle multiple seasonal periods, high-frequency seasonality and non-integer seasonal periods through a trigonometric formulation.

The authors extend the Holt-Winters model with additive seasonality and multiplicative errors in the form of an ETS(M,A,A) model with two seasonal components and incorporate a Box-Cox transformation, ARMA errors as well as up to T seasonal patterns. Livera et al. (2011) propose the following model:

$$y_t = \begin{cases} \frac{y_t^{\omega}-1}{\omega}, & \omega \neq 0 \\ log(y_t), & \omega = 0 \end{cases} \tag{3.59}$$

$$y_t^{\omega} = l_{t-1} + \phi b_{t-1} + \sum_{i=1}^{T} s_{t-1}^{(i)} + d_t \tag{3.60}$$

$$l_t = l_{t-1} + \phi b_{t-1} + \alpha d_t \tag{3.61}$$

$$b_t = (1-\phi)b + \phi b_{t-1} + \beta d_t, \tag{3.62}$$

where Equations (3.59) to (3.62) denote the Box-Cox transformation for observation y_t with parameter ω, the measurement equation, the trend level and the trend growth. The TBATS model also includes a trigonometric formulation of the seasonal component $s_t^{(i)}$ which is given by the following three equations:

$$s_t^{(i)} = \sum_{j=1}^{k_i} s_{j,t}^{(i)} \tag{3.63}$$

$$s_{j,t}^{(i)} = s_{j,t-1}^{(i)} cos\lambda_j^i + s_{j,t-1}^{*(i)} sin\lambda_j^{(i)} + \gamma_1^{(i)} d_t, \tag{3.64}$$

$$s_{j,t-1}^{*(i)} = -s_{j,t-1} sin\lambda_j^{(i)} + s_{j,t-1}^{*(i)} cos\lambda_j^i + \gamma_2^{(i)} d_t, \tag{3.65}$$

where Equation (3.64) describes the stochastic level of the i-th seasonal component, denoted by $s_{j,t}^{(i)}$ and Equation (3.65) describes the stochastic growth $s_{j,t-1}^{*(i)}$ in the level of the seasonal component that can be used to describe the change in the i-th seasonal component over time. The TBATS model employs a Fourier form representation of seasonality here, which allows to model any cyclic function based on seasonal components as a linear combination of trigonometric terms (West & Harrison, 2006). $\gamma_1^{(i)}, \gamma_2^{(i)}$ are seasonal smoothing parameters and k_i is the number of harmonics used to model the i-th seasonal component. Also note that $\lambda_j^{(i)} = 2\pi j/m_i$, where m_i is the i-th seasonal period such that $m_i \in \{m_1, \ldots, m_T\}$.
Finally, the TBATS model also includes an ARMA(p,q) process $\{d_t\}$ to model the autocorrelations found in the white noise error term process $\{\epsilon_t\}$:

$$d_t = \sum_{i=1}^{p} \zeta_i d_{t-i} + \sum_{i=1}^{q} \xi_i \epsilon_{t-i} + \epsilon_t. \tag{3.66}$$

$\sum_{i=1}^{p} \zeta_i$ is the autoregressive component and $\sum_{i=1}^{q} \xi_i$ is the moving average component of the ARMA process of order (p, q). According to Livera et al. (2011), the ARMA error terms are introduced in order to explicitly model the autocorrelation in the error process ϵ_t, which is assumed to be serially uncorrelated in the regular ETS models.

The TBATS model is parametrized as follows:

$$TBATS(\underbrace{\omega}_{\text{BC parameter}}, \underbrace{\phi}_{\text{dampening}}, \underbrace{p,q}_{\text{ARMA parameters}}, \underbrace{\{m_1,k_1\},\{m_2,k_2\},\ldots,\{m_T,k_T\}}_{\text{seasonal period \& \# of harmonics}}), \tag{3.67}$$

where ω is the Box-Cox transformation parameter, which is chosen to carry out the transformation given in Equation (3.59), ϕ is the dampening parameter, which is also used in the regular ETS models. (p, q) denote the order of the ARMA error process. m_i is the i-th seasonal period and k_i is the number of harmonics used for the to describe the i-th seasonal component. $k_i = m_i/2$ for even values of m_i and $k_i = (m_i - 1)/2$ for odd values of m_i.

The estimation of the TBATS model is done via maximum likelihood. Specifically, the following negative **log-likelihood function** is minimized to obtain the maximum likelihood estimates:

$$\mathcal{L}^*(\hat{\boldsymbol{\vartheta}}, \hat{\boldsymbol{x}}_0) = n \log\left(\sum_{t=1}^{n} \epsilon_t^2\right) - 2(\omega - 1) \sum_{t=1}^{n} \log y_t. \tag{3.68}$$

Automated model selection in the TBATS model is carried out by minimizing the AIC

$$AIC = \mathcal{L}^*(\hat{\boldsymbol{\vartheta}}, \hat{\boldsymbol{x}}_0) + 2K, \tag{3.69}$$

where $\hat{\boldsymbol{\vartheta}}$ is an estimate of the vector containing the Box-Cox transformation parameter, the dampening parameter as well as the ARMA process coefficients and $\hat{\boldsymbol{x}}_0$ is an estimate of the initial states vector. K is the total number of parameters in $\boldsymbol{\vartheta}$ plus the number of free states in $\boldsymbol{x}_0$.

Furthermore, the number of harmonics is not selected by evaluating every possible combination but rather through an approach which involves the approximation of the seasonally de-trended data with a regression and then gradually adding harmonics and testing for their significance until a model fitted to the data with a minimal AIC emerges.

Moreover, Livera et al. (2011) state that a two step-procedure is employed to determine the order (p, q) of the ARMA process according to which a suitable model with no ARMA component is selected first and subsequently the Hyndman-Khandakar algorithm is applied to the residuals of said model in order to determine p and q. The selected first model is then fit again augmented by the ARMA errors components but the latter are only retained if the new augmented model leads to a lower AIC than the model without the ARMA component.

According to Livera et al. (2011), a further advantage of the TBATS model is that it can also be used to decompose a complex seasonal series into the trend, seasonal and irregular components.

A further model variation, which can also be considered as an extension of the ETS model to accomodate more complex seasonal patterns, is the **Exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal components (BATS)**. The BATS model can also handle up to T seasonal patterns but it models the seasonal patterns in a different way. Equations (3.63) to (3.65) are replaced by the following formulation for each individual seasonal component:

$$s_t^{(i)} = s_{t-m_i}^{(i)} + \gamma_i d_t, \tag{3.70}$$

where $\gamma_i, i = 1, \ldots, T$ are seasonal smoothing parameters. Livera et al. (2011) state that the BATS model comes with the limitations that it cannot handle non-integer seasonal periods and that the number of seed states can grow very large in case of seasonal patterns with high seasonal periods.

3.3 Ensemble Methods

This section describes the statistical learning method of ensembling. The most recent advances with respect to a time series context are also described, when applicable. Hastie, Tibshirani, and Friedman (2009) describe **ensemble learning** as the two step process of developing a collection

of weaker base models, called "base learners", from the training data and then subsequently combining them to form a composite prediction model. Murphy (2012) formulates this weighted combination of base models as follows:

$$f(y|\boldsymbol{X}) = \sum_{m \in \mathcal{M}} w_m f_m(y|\boldsymbol{X}), \tag{3.71}$$

where w_m are the ensemble weights and f_m are the base models.
Generally, there are three basic techniques that can be described by the overarching term of ensemble methods, which are bootstrap aggregating (bagging), boosting, and stacking. In this paper, we concentrate on the first two techniques and highlight advances in the domain of times series forecasting.

3.3.1 Bagging for Time Series

Bootstrap aggregation (Bagging) is a general-purpose technique for reducing the variance of a statistical learning method, which can also increase that method's prediction accuracy (James et al., 2013).
In order to obtain an overall prediction from a bagging ensemble model, which we refer to as the bagging estimate, we first have to draw B bootstrap data sets with replacement from the training data: $\{(\boldsymbol{x}_b^{(i)}, y_b^{(i)}); i = 1, \ldots, n\}$. Then, we predict $\hat{f}_b(\boldsymbol{X})$ based on the b-th bootstrap data set for $b = 1, \ldots, B$. Finally, according to Hastie et al. (2009), the bagging estimate or the prediction from the ensemble is obtained as follows:

$$\hat{f}^{bag}(\boldsymbol{X}) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}_b(\boldsymbol{X}). \tag{3.72}$$

The well-established bagging method was first introduced by Breiman (1996) but it was not successfully applied in a time series forecasting context until 2016.
According to Petropoulos et al. (2018) the complication with time series consists in accounting for non-stationarity and autocorrelation in the bootstrapping procedure in order to produce bootstrapped samples that resemble the original data.
Bergmeir, Hyndman, and Benıtez (2016) propose a bootstrapping procedure for time series as illustrated in Figure 3.2 that includes a Box-Cox transformation in order to stabilize the variance and in order to ensure that the components of the series are additive. The Box-Cox transformation, which was first introduced by Box and Cox (1964), is defined as follows:

$$w_t = \begin{cases} log(y_t), & \omega = 0 \\ (y_t^{\omega} - 1)/\omega, & \omega \neq 0 \end{cases}. \tag{3.73}$$

The optimal $\omega \in [0, 1]$ is chosen by dividing the series into subseries of length equal to the seasonality and minimizing the coefficient of variation $\sigma/\mu^{(1-\omega)}$ across those subseries, where σ stands for the standard deviation and μ for the sample mean of the subseries.
Subsequently, a decomposition either in form of the loess method to extract trend and remainder in case of a non-seasonal time series or in form of a STL decomposition in order to break a seasonal series down into the trend, seasonal and remainder components. The remainder of the decomposed series is then assumed to be stationary but there may be autocorrelation.
The authors then apply a bootstrapping method that allows for autocorrelation, the Moving block bootstrap (MBB) in a slightly modified version of the original method suggested by Künsch (1989), to the extracted remainder of the series. De Oliveira and Cyrino Oliveira (2018) provide further details on the MBB method, which they also adopt. They state that the proposed MBB approach consists of drawing $(n/l) + 2$ overlapping blocks from the remainder of the series and then discarding a random number between 0 and $l - 1$ from the beginning of the bootstrapped series and a further number of values to obtain the same length as the original remainder series, which is bootstrapped. n is the length of the original series and l is the block size. In this modified version, any value from the original series can possibly be placed anywhere in the bootstrapped series. In the next step, the series is reconstructed from its structural components, i.e. trend, seasonality, and the bootstrapped remainder. Finally, the Box-Cox transformation is inverted. The whole process is then repeated in order to obtain multiple bootstrapped series.
Bergmeir et al. (2016) state that their ensemble bagging model is then created by applying the *ets()* R function to each of the bootstrapped series, which autoselects the ETS model that minimizes the AIC. The point forecasts from all these models are then combined using the median. Note that this bagging procedure can also be applied to models other than ETS and the point forecasts from the base models can also be combined using a different statistical measure such as the arithmetic mean.

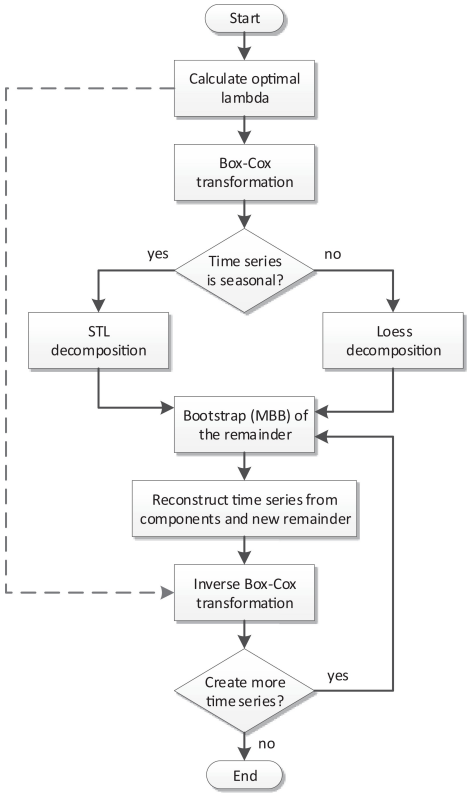

Figure 3.2: Bootstrapping procedure for a time series

Notes: This figure is taken from Petropoulos et al. (2018) and shows a flowchart of the bootstrapping procedure for a time series developed by Bergmeir et al. (2016). This procedure involves a Box-Cox transformation to stabilize the variance, which is followed by a decomposition of the time series into the remainder and the trend component as well as a seasonal component if the latter is present in the series. A moving block bootstrapping procedure, which allows for autocorrelation, is then applied to the remainder of the series. The remainder is assumed to be stationary. Finally, the other components are combined with the bootstrapped remainder and the Box-Cox transformation is inverted to reconstruct the time series. This process is then repeated to create multiple such reconstructed series which are called the bootstraps.

Bergmeir et al. (2016) find that the ensemble of bagged exponential smoothing models outperforms the regular exponential smoothing model consistently for monthly data on the M3 forecasting competition data set, which is a common medium of comparison of newly introduced forecasting methods with existing state of the art models.

However, the most common base model across all ensemble methods is a decision tree. Moreover, it is also possible to refine the bagging prediction technique and reduce the variance even further under certain circumstances and therefore we first introduce the concept of a decision tree in order to explain the concept of a Random Forest.

In general, the purpose of **decision tree** based methods, which are typically used for regression

or classification problems, is to segment the feature space, which can also be named the predictor space, into a number of simpler regions. The name stems from the fact that the set of splitting rules can be described with a tree (James et al., 2013). For the purpose of this paper, we will focus on **regression trees**.
Figure 3.3 illustrates the tree resulting from the partition of a two-dimensional feature space and also depicts the prediction surface resulting from that tree.

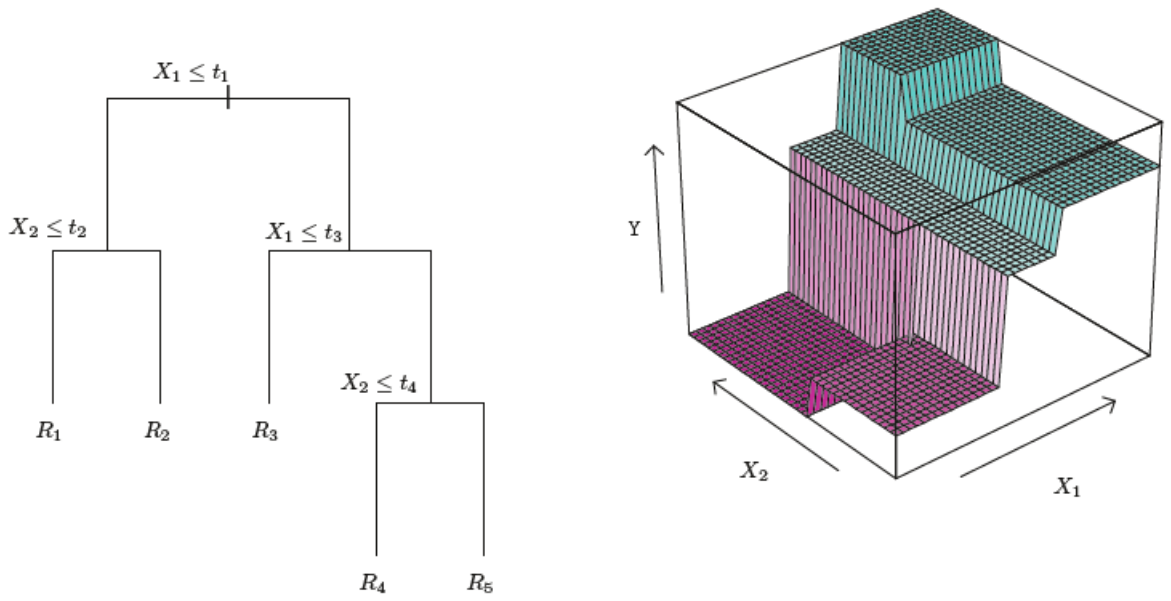

Figure 3.3: Decision tree and resulting prediction surface

Notes: This figure taken from James et al. (2013) is composed of two panels. The left panel shows the decision tree resulting from the partition of a two dimensional predictor space, i.e. the set of possible values for predictors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, into 5 Regions $R_1, \ldots, R_5$, where $t_1, \ldots, t_4$ denote the cutting points for binary splitting. Note that t_1 is referred to as a parent node and t_2, t_3 and t_4 are referred to as child nodes or simply childs. The final five regions of the predictor space are also referred to as terminal nodes. The right panel shows the prediction surface associated with this tree as values for the two predictors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are varied.

The regions $R_j, j = 1, ..., 5$ are called **terminal nodes** or **leaves**, the points $t_i, i = 1, ..., 4$ where the predictor space, i.e. the set of possible values for predictors $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_p$, is split are referred to as **internal nodes** and the connecting segments are called **branches**.
More formally, incorporating the notation from John and Townshend (2019), we intend to partition the feature space $\mathcal{X}$ into J disjoint regions as follows:

$$\begin{aligned} &\mathcal{X} = \cup_{j=0}^{J} R_j \\ \text{s.t.} \quad &R_j \cap R_i = \emptyset \, \text{for} \, j \neq i, \end{aligned} \tag{3.74}$$

where $J \in \mathbb{Z}^+$.

James et al. (2013) argue that the goal of a regression tree is to split the feature space $\mathcal{X}$ into J distinct and non-overlapping regions $R_1, R_2, ..., R_J$ in such a way that the Residual sum of squares (RSS) is minimized as follows:

$$\min_{R_1,\dots,R_J} \quad RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2, \tag{3.75}$$

where $\hat{y}_{R_j}$ is the mean response for the training observations in the j-th region. More precisely, the final prediction for region R_j is the mean of all y_i falling into the specific region, which can be expressed as

$$\hat{y}_{R_j} = \frac{\sum_{i \in R_j} y_i}{|R_j|} = \gamma_j. \tag{3.76}$$

Another key ingredient of decision trees is the concept of "recursive binary splitting", which refers to a top-down approach to successively splitting the predictor space. Mathematically, the predictor $\boldsymbol{x}_j$ and the cutting point t are chosen such that the split of the predictor space into the regions $\{\mathcal{X}|\boldsymbol{x}_j < t\}$ and $\{\mathcal{X}|\boldsymbol{x}_j \geq t\}$ leads to the maximum reduction in the RSS.

A **Random Forest** provides a way of improving bagged decision trees by decorrelating the trees through a tweak. Predictions from bagged trees will be highly correlated if there are a very strong predictor and several moderately strong predictors in the dataset. In order to resolve this problem, the Random Forest method can improve the reduction in variance by modifying the tree growing process in the following way: Whenever a tree is grown on a bootstrap dataset, only $m \leq p$ of the input variables are considered at random for a split to be made. The reduced design matrix is denoted $\tilde{\boldsymbol{X}}$. Typically, $\sqrt{p}$ is the number chosen for m (Hastie et al., 2009). Formally, the random forest predictor for B trees grown in this modified way is

$$\hat{f}^{rf}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}_b(\tilde{\boldsymbol{X}}), \tag{3.77}$$

where the function $\hat{f}_b(\tilde{\boldsymbol{X}}) = T(\tilde{\boldsymbol{X}}; \Theta_b)$ is the general formal parametric representation of the b-th decision tree, which in this case forms a base model to obtain a random forest predictor.[3] The parameters $\Theta_{b \in \{1,\dots,B\}} = \{R_j^*, \hat{y}_{R_j}; j = 1, \dots, J\}$ are found by minimizing the empirical risk function and they characterize each decision tree. $R_j^*; j = 1, \dots, J$ denote the final regions of the predictor space that minimize the RSS. $\hat{y}_{R_j}; j = 1, \dots, J$ represent the respective predictions per region.

[3] For further details see Hastie et al. (2009, p. 356).

Nevertheless, the literature on time series random forests that can be applied to regression tasks is very sparse at this point in time. However, Deng, Runger, Tuv, and Vladimir (2013) propose a time series forest that can be used for classification tasks. High quality splits are obtained by considering the entrance gain, a measure combining the entropy gain and a distance measure. The authors also state that the computational complexity is linear in the length of the time series and that a temporal importance curve consisting of entropy gains at different time indices can be used to capture important interval features such as the median for a specific interval of the underlying time series.
Due to a lack of availability of refined Random Forest regression methods for time series, we will stick to the regular bagging technique for the purpose of this paper.

3.3.2 Boosting for Time Series

Another ensemble method based on regression trees is **gradient tree-boosting**, the goal of which is to sequentially grow trees using information from previously grown trees. In contrast to the simultaneous procedure of bagging, no bootstrapping of the training data is required. Instead, trees are grown using the current residuals rather than $\boldsymbol{y}$ as a target variable. After every iteration, the last fitted tree is added to the current boosted tree ensemble model in order to update the residuals.
The boosted tree model is

$$f_{\mathcal{M}}(\boldsymbol{X}) = \sum_{m=1}^{\mathcal{M}} f_m(\boldsymbol{X}), \tag{3.78}$$

where $f_m(\boldsymbol{X}) = T(\boldsymbol{X}; \Theta_m)$ denotes the tree structure of the m-th base learner. According to Hastie et al. (2009), the algorithm starts by initializing the optimal constant model,

$$f_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma), \tag{3.79}$$

which represents a tree with a single terminal node, region R with the associated prediction $\hat{y}_R = \gamma$. Then, the generalized residuals r_{im}, which represent the target values to fit the regression trees to, are computed for the training data $i = 1, ..., n$ by using the negative gradient evaluated at f_{m-1}

$$r_{im} = -[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}]|_{f=\boldsymbol{f_{m-1}}}. \tag{3.80}$$

At each iteration m, we fit one regression tree to the targets r_{im} which yields the terminal regions R_{jm} for $j = 1, ..., J_m$. J_m represents the size of each single base model tree included in the boosting ensemble. In total, we fit $\mathcal{M}$ regression trees over the course of $\mathcal{M}$ iterations. J_m

and $\mathcal{M}$ are the main parameters to be tuned in the gradient tree-boosting algorithm.

After computing the optimal constant per region, γ_{jm} for $j = 1, ..., J_m$, which is equivalent to the mean of all y_i falling into that specific region, the solution is updated in a forward additive stage-wise boosting procedure. We augment the generic gradient tree-boosting procedure as described by Hastie et al. (2009) by a shrinkage parameter η such that $0 \leq \eta \leq 1$, which is equivalent to the learning rate used in the gradient descent algorithm for neural networks given in Equation (3.115). The gradient boosting algorithm then updates the initial prediction $f_0(x)$ and every successive $f_m(x)$ is based on the current parameter vector $\boldsymbol{f_{m-1}}$, which is a sum of the previously incorporated updates. Adapting the notation used by Friedman (2002) to our paper leads us to the following update equation per boosting iteration m:

$$f_m(\boldsymbol{X}) = f_{m-1}(\boldsymbol{X}) + \eta * T(\boldsymbol{X}; \Theta_m). \tag{3.81}$$

Finally, the predicted output of the gradient-boosted tree model is obtained after $\mathcal{M}$ iterations as $\hat{f}^{boost}(\boldsymbol{X}) = f_{\mathcal{M}}(\boldsymbol{X})$.

According to Hastie et al. (2009), boosting is among the most powerful statistical learning techniques introduced in the recent two decades. When it comes to a time series regression context, Barrow and Crone (2016) state that seven variants of the Adaboost algorithm have been extended to regression tasks with time series data. In its original form, the **Adaboost** algorithm produces a weighted composite prediction combining the binary predictions of a sequence of iteratively trained classifiers, which are applied to sequentially updated weighted versions of the original data sample. The weights are also computed by the boosting algorithm and a higher weight is given to previously miss-classified observations of the sample which forces the successive classifier to focus on the observations that were missed by the previous classifier.

According to Barrow and Crone (2016), the most common base models to boost using Adaboost algorithm variants in the context of time series data are multi-layer perceptrons, which are explained in more detail in Section 3.4.1 and regression trees. Barrow and Crone (2016) evaluate the forecasting performance on the NN3 competition dataset consisting of 111 time series and find that regression trees as base models are outperformed by multi-layer perceptrons across all data conditions for all boosting variants. However, they also find that even the best boosting variant using multi-layer perceptrons as a base learner performs worse at time series prediction than a bagging ensemble.

Another highly successful boosting variant is **Extreme gradient boosting (XGB)**, which

describes a scalable Machine Learning system for decision tree boosting that has first been developed by Chen and Guestrin (2016). This machine learning method is widely used by data scientists and Chen and Guestrin (2016) argue that its successful application mainly stems from its scalability due to parallel and distributed computation capabilities as well as due to algorithmic approximations to facilitate handling sparse data and approximating tree learning. An approximate algorithm to find a good tree split can be necessary if carrying out the exact greedy algorithm, which tries to find the optimal split by iterating over all possible splits over all the features, would be computationally impossible due to memory restrictions.

Chen and Guestrin (2016) modify the objective function by introducing a regularization term $\Omega(f_m)$ in order to prevent overfitting :

$$\mathcal{L}(L(\cdot), \Omega(\cdot)) = \sum_i L(y_i, \hat{y}_i) + \sum_m \Omega(f_m), \tag{3.82}$$

where $L(\cdot)$ is a differentiable convex loss function based on the prediction $\hat{y}_i$ of the target y_i over all data points. f_m is the m-th regression tree which forms a base model for the tree boosting ensemble model that corresponds to a specific tree structure and specific leaf weights. We set up the regularization function for an individual tree in such a way that it allows for the inclusion of a ℓ_1 as well as a ℓ_2 regularization term as follows:

$$\begin{aligned} \Omega(f_m) &= \gamma T + \alpha \underbrace{\|w\|_1}_{\ell_1 \text{ penalty}} + \frac{1}{2}\lambda \underbrace{\|w\|_2^2}_{\ell_2 \text{ penalty}} \\ &= \gamma T + \alpha \sum_i \mid w_i \mid + \frac{1}{2}\lambda \sum_i w_i^2, \end{aligned} \tag{3.83}$$

where w denotes the sum over the continuous leaf weights w_i and T is the number of leaves in a tree. Note that the objective function is identical to that of the regular gradient tree-boosting method when the ℓ_1 regularization parameter $\alpha \in [0, \infty[$ as well as the ℓ_2 regularization parameter $\lambda \in [0, \infty[$ are set to 0.

Moreover, the XGB method leverages two more techniques to prevent overfitting, shrinkage and column subsampling. Chen and Guestrin (2016) state that the function of shrinkage is to scale newly added weights with a factor of η after each step of tree boosting in order to allow future trees to improve the model further. Column subsampling refers to the fraction of observations to be randomly considered for fitting each tree, which is the technique that is also used in Random Forests.

The most recent gradient tree boosting model algorithm, **Light gradient boosting machine**

(LGBM), which has been proposed by Ke et al. (2017), extends the scalability to high feature dimensions and large data sets even further than the XGB algorithm does. The authors argue that their LGBM algorithm can speed up the training process by a factor of up to 20 compared to common gradient tree boosting algorithms while maintaining the accuracy of comparable algorithms. LGBM also supports parallel processing.
Ke et al. (2017) state that they tackle the problem of extensive computational requirements when estimating the information gains on all possible split points for every single feature by introducing two new techniques: gradient based one side sampling and exclusive feature bundling. The first technique serves the purpose of estimating an accurate information gain with a much smaller data size since a part of the data instances with small gradients is excluded from this computation. The second technique can be used to reduce the number of features by bundling mutually exclusive features without compromising on the ability to detect accurate splitting points of the predictor space.
LGBM accomodates the same hyperparameters to prevent overfitting as XGB does and we will therefore implement the LGBM boosting algorithm due to its computational speed advantages compared to the XGB algorithm.

3.4 Neural Network Models

Artificial neural networks (ANNs) refer to mathematical models which are based on the biological structure of the brain and which allow for complex non-linear relationships between a label and features. Neural networks can have various different network architectures which may be advantageous for different specific applications.[4]
This section introduces the structure of an artificial neuron as a building block of neural networks. Next, different network architectures are presented and it is explained how these are used as functional approximators. First, feedforward neural networks (FFNNs) are explained, then recurrent neural networks (RNNs) and in particular two RNN architecture variants, the Long Short Term Memory (LSTM) and the Gated Recurrent Unit (GRU) neural networks are presented. Lastly, we explain how these neural networks can be employed in the domain of time series forecasting, how prediction intervals can be constructed and how their accuracy can be evaluated.

3.4.1 Functioning Principles and Building Blocks

ANNs are a class of algorithms within machine learning that is based on the mathematical disciplines but it is also uniquely inspired by neuroscience (Haykin, 1999). The first references

[4]See (Bengio et al., 2009) for a holistic overview of the different architectures for deep learning models.

of neural networks in the literature go back to the 1940s, when cognitive computing based on brain-like structures were first conceptualized (McCulloch & Pitts, 1943). The underlying motivation was that the human brain is able to process and solve immensely complex problems that are beyond the reach of any computer. This is even more remarkable due to the fact that a human brain is a relatively small but highly efficient processor.

Although many of the brain's inner workings are not yet known to scientists, we do know a fair amount of its structure. The brain consists of approximately 10 billion neurons, which are cells unique to the brain. The minimal structure of biological neurons which is adopted by artificial neurons used in computing are dendrites, which are transmission channels for incoming information, the cell body, and axons, which transmit output signals. Furthermore, each neuron is connected with around 10,000 other neurons trough contact points between different neurons, which are called synapses, that send signals back and forth. These synapses have the ability to regulate the strength of the signal coming trough it based on previous experience. The neural cell processes the sum of weighted inputs from other connected cells and if this measure reaches a certain threshold, the cell is activated (Haykin, 1999). Synapses therefore have weights associated to them which are referred to as synaptic weights.

ANNs are an attempt at modeling the information processing capabilities of nervous systems, the building blocks of which are called neural cells or neurons (Rojas, 1996). Weights are also used in artificial neurons and are associated with the connections between a node and the input and output channels. A node represents the cell body in an artificial neuron.
The building block of an artificial neural network is called a perceptron which is mathematical model of a biological neuron with the aforementioned four components. A graphical illustration is presented in Figure 3.4. Note that perceptrons can either have one or multiple outputs.

According to Vlahogianni et al. (2004), artificial neural networks are especially suitable for time series forecasting since they are non-parametric, i.e. they do not require any stationarity assumptions of the time series used as an input. Moreover, due to their extensive mapping capabilities, ANNs are more flexible than more traditional statistical models such as ARIMA models (Yan, 2012). However, the enormous flexibility and lack of assumptions that neural networks provide come at a cost. Due to the fact that neural networks approximate their target functions through a supervised learning algorithm instead of the estimation of function parameters, they have been labeled a "black box" algorithm and they provide little insight into the relative influence of the features in the prediction process (Sussillo & Barak, 2013).

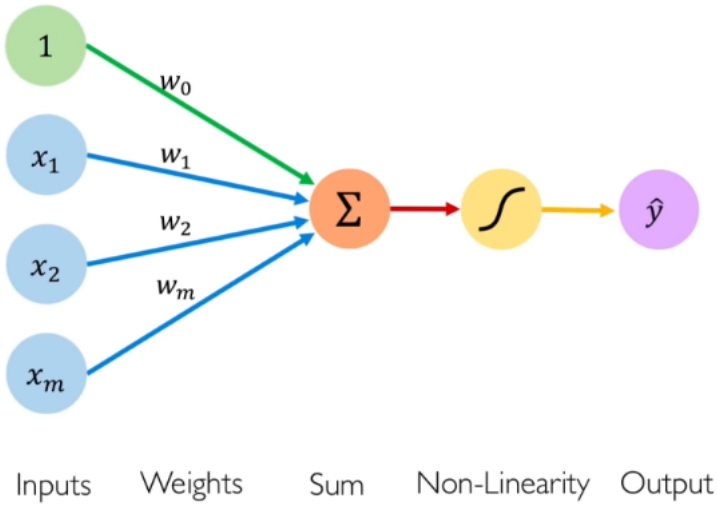

Figure 3.4: Structure of a perceptron or single artificial neuron

Notes: This figure taken from Amini and Soleimany (2020a) shows the structure of a single output perceptron. The inputs $x_1, x_2, \ldots, x_m$ are multiplied with their associated weights $w_1, w_2, \ldots, w_m$ and are then summed up to form a linear combination. This combination of inputs is then passed forward through a nonlinear activation function $g(\cdot)$, which is represented by the yellow symbol to produce the output $\hat{y}$. The perceptron also contains a bias term w_0 which is shown in green.

The perceptron can be formulated by the following single equation

$$\hat{y} = g(w_0 + \sum_{i=1}^{m} x_i w_i) \tag{3.84}$$

or we can make the notation even more compact by using linear algebra and applying the dot product, which leads us to

$$\hat{y} = g(w_0 + \boldsymbol{x}^T \boldsymbol{w}), \tag{3.85}$$

where $\boldsymbol{x}^T = (x_1, \ldots, x_m)$ and $\boldsymbol{w}^T = (w_1, \ldots, w_m)$. Note that $\boldsymbol{x}, \boldsymbol{w} \in \mathbb{R}^m$.
The activation function $g(\cdot)$ can be modeled in various different ways depending on the application of the neural network. More specifically, $g : \mathbb{R} \to \mathbb{R}$ is a differentiable non-linear mapping that is applied elementwise to the vector-valued activation, which is the linear combination of weights w_i and inputs x_i. Note that w_0 denotes the bias parameter. This term has the purpose of shifting the activation function irrespective of the inputs. Amini and Soleimany (2020a) state that the main purpose of an activation function is to introduce non-linearities into the neural network which allows for the approximation of arbitrarily complex functions.

Common activation functions are shown in Figure 3.5.

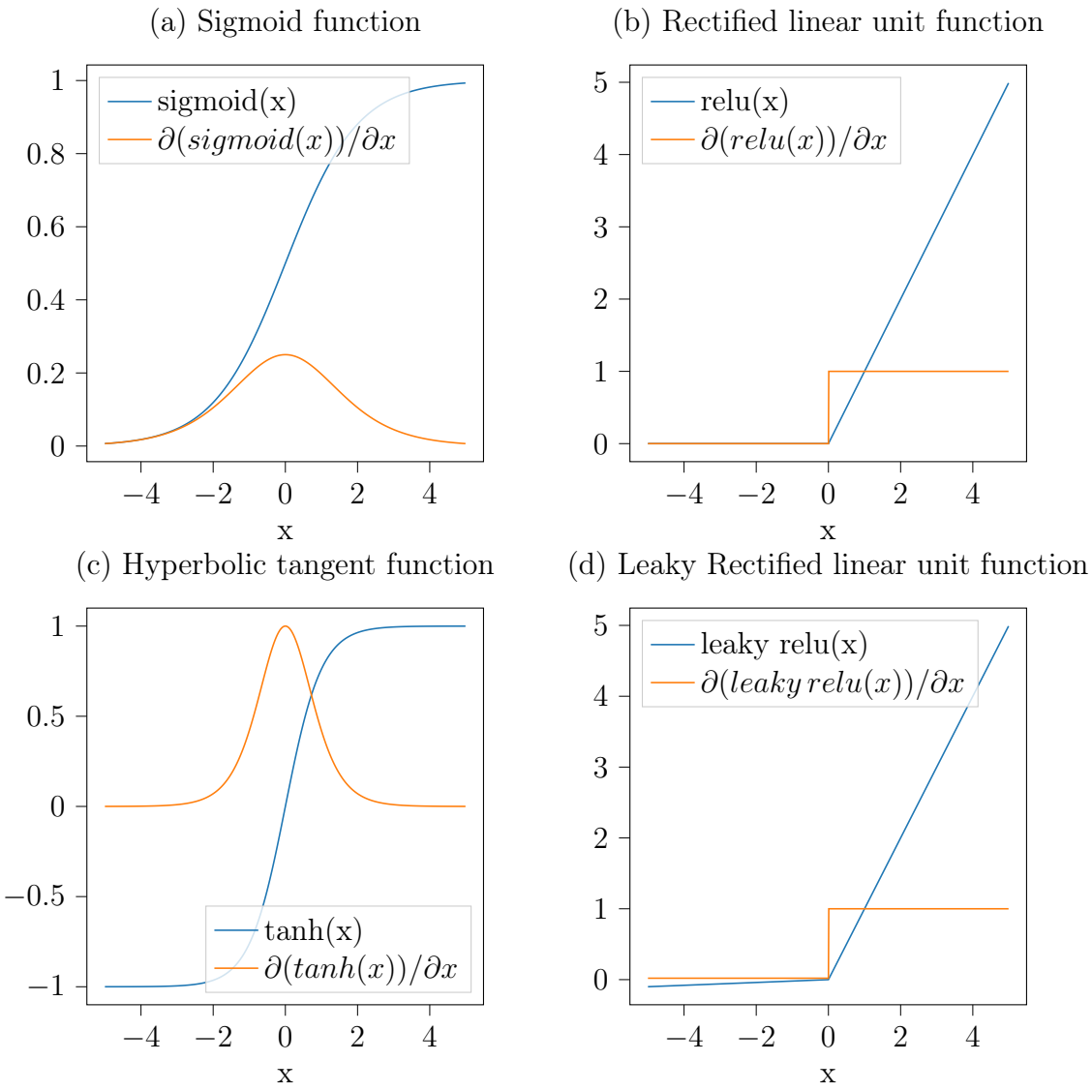

Figure 3.5: Graphs of Common Activation Functions and their first derivatives

Notes: Panel (a) shows the sigmoid activation function, panel (b) depicts the rectified linear unit (RELU) activation function, panel (c) shows the hyperbolic tangent activation function and panel (d) shows the leaky RELU activation function with a slope $\delta = 0.02$ for $x < 0$. The respective equations are given in Equations (3.86) to (3.93).

These include the **sigmoid** function,

$$sigmoid(x) = (1 + exp(-x))^{-1}, \tag{3.86}$$

which only outputs values in the codomain of $[0, 1]$. The first derivative with respect to the argument x is given by

$$\partial(sigmoid(x))/\partial x = \frac{exp(-x)}{(1+exp(-x))^2}. \tag{3.87}$$

A further popular activation function is the **Rectified linear unit (RELU)**, which is defined as

$$relu(x) = \max_x(x, 0) \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}. \tag{3.88}$$

The RELU function is linear for positive values and outputs zero for negative values which can help to avoid the vanishing gradient problem explained in Section 3.4.4 which can occur when using the sigmoid or the hyperbolic tangent function and it also means that neurons get sparsely activated. The first derivative of the RELU activation function is given by

$$\partial(relu(x))/\partial x = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}. \tag{3.89}$$

A further relevant activation function which is used for the more sophisticated neural network architectures of the LSTM and the GRU is the **hyperbolic tangent** function, which is is defined as

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{3.90}$$

Moreover, the first derivative of the tanh function is

$$\partial(tanh(x))/\partial x = 1 - tanh(x)^2. \tag{3.91}$$

Lastly, the **leaky RELU** activation function modifies the regular RELU function in such a way that it allows for small negative values when the input is less than zero. According to Brownlee (2018), the leaky RELU function can be used to overcome the "dying RELU" problem , which means that if a neuron using a RELU activation function gets stuck in a situation of always outputting a value of 0, that node never activates. In that case the gradient is equal to 0 as well and the weights for that neuron are not updated. Since the leaky RELU function has a non-zero slope for negative inputs, it can avoid this problem. The leaky RELU function can be written as follows:

$$leaky\, relu(x) = \max_x(\delta x, x) = \begin{cases} \delta x, & x < 0 \\ x, & x \geq 0 \end{cases}, \tag{3.92}$$

where δ is the slope parameter for negative arguments. The first derivative with respect to the argument x is

$$\partial(leaky\,relu(x))/\partial x = \begin{cases} \delta, & x < 0 \\ 1, & x \geq 0 \end{cases}. \tag{3.93}$$

3.4.2 Feed Forward Neural Network (FFNN)

Feed-forward neural networks (FFNN) are comprised of three main elements. Neurons, layers, wherein the neurons are located, and the connections between neurons. Neurons will interchangeably be referred to as nodes and the connections between neurons will be referred to as weights. The network architecture is then built up by sequentially chained layers where the weights connect each layer to the next one.

When considering feedforward neural networks, the architecture can be parametrized by k, the number of hidden layers, and by n_k, the number of neurons in a given layer k.
The first layer in the network is called the **input layer** and here the neurons contain the features of the data, $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_m\}$, each of which contain n observations. We define the output of the j-th neuron in a given layer L by $\boldsymbol{z}_j^{(L)}$. The output of a neuron in the input layer is simply the feature-vector associated with that neuron, $\boldsymbol{z}_j^{(1)} = \boldsymbol{x}_j$.
All of the following layers except for the very last one are called **hidden layers**. In these layers, the output of each neuron is computed in two steps. First, a linear combination of weights and inputs, i.e. the activation is calculated. The inputs are simply the last layer's outputs $\boldsymbol{z}^{(L-1)}$ and the weights are a set of parameters $w_{i,j}^{(L)} \in \boldsymbol{W}^{(L)}$ denoting the weight between neuron i in layer L and neuron j in layer $L+1$. $\boldsymbol{W}^{(L)}$ is called the weight matrix associated with layer L. Second, the non-linear activation function $g(\cdot)$ is applied to produce the output of the hidden neuron $\boldsymbol{z}_i$.
Finally, the last layer of the network is called the output layer and its output is computed in an identical fashion to that of the hidden layers. The purpose of the activation function in this case is no longer the introduction of non-linearities but rather the scaling of the final output of the model to its desired range. The last layer of the network represents the combined output of the model, $\hat{f}(\boldsymbol{X}; \boldsymbol{W}) = \hat{\boldsymbol{y}} = \boldsymbol{z}^{(k+1)}$. This is a non-linear mapping of the input $\boldsymbol{X} \in \mathbb{R}^{n \times m}$ to the output $\hat{\boldsymbol{y}} \in \mathbb{R}^{n \times h}$. h is the dimension of the output. $\hat{f}_{\boldsymbol{W}}(\cdot)$ is also referred to as the **network function** and is parametrized by the set of weights combining each layer, $\boldsymbol{W} = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, ..., \boldsymbol{W}^{(k+1)}\}$. It is by adjusting these weights that the network can adapt the features given in the input layer to the target in the output layer.
Figure 3.6 shows a single hidden layer FFNN with two outputs.

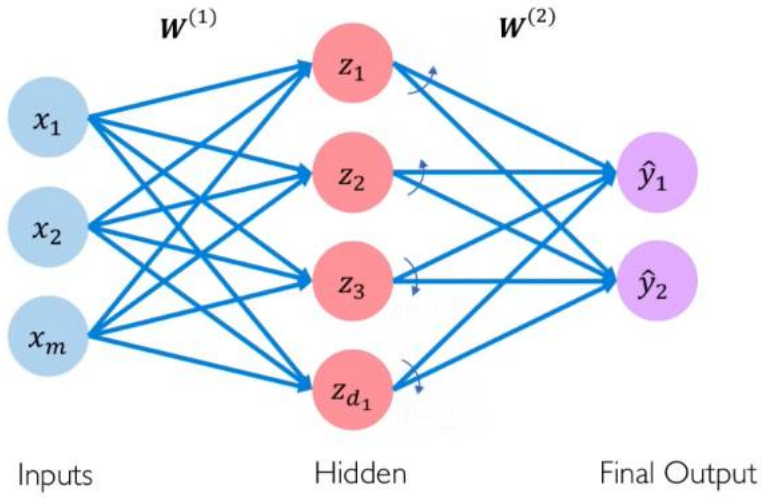

Figure 3.6: Multilayer feed-forward network with one hidden layer

Notes: This figure is taken from Amini and Soleimany (2020a) and shows a dense single layered FFNN with a single input layer and a single hidden layer which feeds into a single output layer. There are two weight matrices, $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ - one for connections between input and hidden layer and one for connections between hidden and output layer. Note that the red colored nodes represent hidden nodes and each hidden node represents a single output perceptron which performs the transformation from Equation (3.84) and passes the output on to the next layer. In the illustrated network there are m inputs, d_1 hidden nodes and 2 outputs. Bias terms are omitted in this illustrative example.

The two transformations underlying the FFNN illustrated in Figure 3.6 can be represented by two equations. The transformation between the inputs and a single hidden node, i.e. the **activation**, a_i for $i = 1, \ldots, d_1$ in layer 1 is

$$a_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \tag{3.94}$$

and the transformation between the input layer and the hidden layer produces the ouput of the hidden neuron

$$z_i = g(a_i) = g(w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}). \tag{3.95}$$

Between the hidden layer and the output layer another transformation of the following form takes place:

$$\hat{y}_i = g(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(a_j) w_{j,i}^{(2)}). \tag{3.96}$$

The bias parameters $w_{0,i}^{(1)}, w_{0,i}^{(2)}$ and weight parameters $w_{j,i}^{(1)}, w_{j,i}^{(2)}$ are learned from the data, but

the weights are initially set to random values and are then updated from the observed data. According to R. J. Hyndman and Athanasopoulos (2018), the number of hidden layers and the number of neurons contained in the hidden layers is usually determined in advance by cross-validation.

Deep learning refers to neural network architectures with more than one hidden layer. For example, a graphical representation of a deep FFNN with k hidden layers can be seen in Figure 3.7.

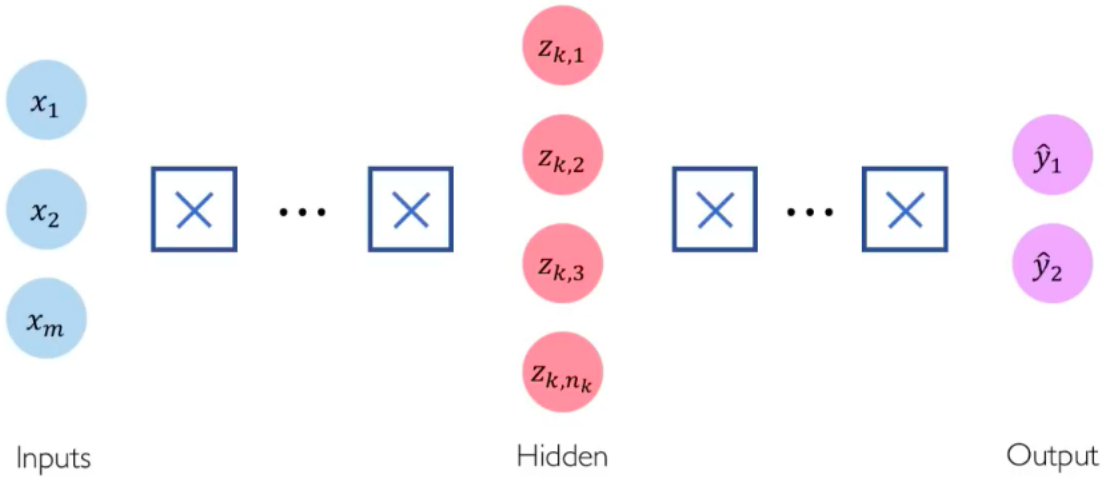

Figure 3.7: Deep feed-forward network with k hidden layers

Notes: This figure taken from Amini and Soleimany (2020a) shows a deep FFNN with k hidden layers and n_k hidden neurons, whereas the number of inputs is still m and the number of outputs is still 2 just as in Figure 3.6. Note that the symbols in between the inputs and the hidden neurons as well as in between the hidden neurons and the output represent fully connected dense layers.

When a feed-forward neural network takes an input feature vector $\boldsymbol{x}$ in order to produce the output $\hat{\boldsymbol{y}}$, information flows forward through the feed-forward network. The initial information provided by the input then propagates through the hidden units of each layer and finally produces the output. This process described by Goodfellow, Bengio, and Courville (2016) is called **forward propagation**. The training of the feed-forward network starts with the initialization of all weights to small randomily chosen values and the initialization of the bias terms to zero or small positive values. Then, the forward propagation is carried out until the mininal scalar value of the cost function $J(\boldsymbol{\theta})$ is found, where the cost function with respect to the training set can be written as given by Goodfellow et al. (2016) as follows:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y), \tag{3.97}$$

where $\hat{p}_{data}$ is the empirical distribution and $f(\boldsymbol{x}; \boldsymbol{\theta})$ is the predicted output from the use of $\boldsymbol{x}$ as an input vector. y is the true value of the output. By comparing the predicted output with the true value of the output and by taking the conditional expectation, a value of the total cost incurred by the neural network can be computed using the pre-specified per training example loss function for the network.
Information in feed-forward neural networks only flows in one direction and there are no cycles or loops in contrast to recurrent neural networks which are explained in Section 3.4.3.
The network is then trained by minimizing the set of weights, $\boldsymbol{W}$, that minimizes the loss function using either the regular stochastic gradient descent with a fixed learning rate or another learning algorithm which involves an adaptive learning rate. The learning rate refers to scalar number with which the weights are updated in the training process of a neural network. Finally, the gradients of the loss function with respect to every weight in the network are calculated through backpropagation from the output to the input of the network. Details on the learning algorithm and on the backpropagation algorithm are given in Section 3.4.6.

3.4.3 Recurrent Neural Network (RNN)

Recurrent neural networks (RNNs) come with the distinct characteristic feature of a recurrent cell which allows for information to persist over time by passing it on internally within the network from one time step to the next one in contrast to regular ANNs. This is done through a single component of the cell state called the hidden state $\boldsymbol{h}_t$.
Hewamalage, Bergmeir, and Bandara (2019) state that RNNs are the most common architecture for sequence prediction problems since the feedback loops of the recurrent cells can be used to address temporal dependencies and the temporal order of sequences. The architecture of a RNN cell as well as the set of recurrent computations is shown in Figure 3.8.
The forward propagation differs from that of the FFNN in that the forward pass from input to output is done at every time step and a loss is computed. The backpropagation is also different from that carried out in the case of a FFNN because the loss has to be backpropagated at every time step as well as across all time steps to the beginning of the sequence. Thus, the gradients of each time step specific loss are computed with respect to the respective weights and then the weight parameters are updated in order to the train the RNN by minimizing the pre-specified loss function.
Goodfellow et al. (2016) argue that this neural network architecture with recurrent connections between hidden nodes is very powerful but also very memory cost and runtime intensive since forward propagation is inherently sequential and since states computed during the forward pass have to be stored until their reusage during the backward pass.

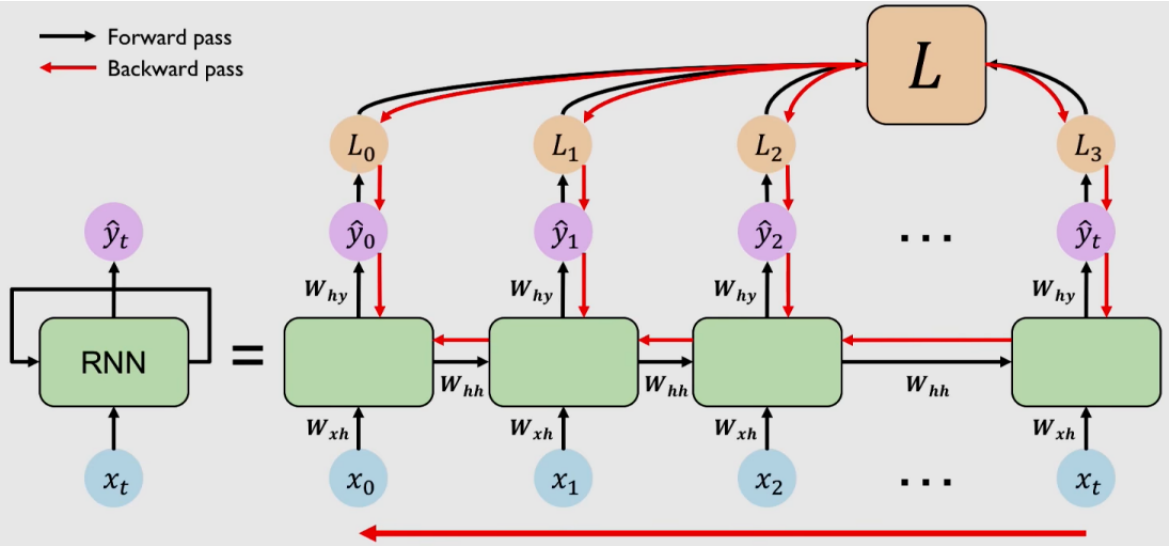

Figure 3.8: RNN computational graph

Notes: This figure, which is taken from Amini and Soleimany (2020b), shows the structure of a RNN neural network cell and the set of recurrent computations as a computational graph, where $\boldsymbol{x}_t$ denotes the input vector, $\boldsymbol{h}_t$ denotes the current hidden state. This figure illustrates how RNNs can be thought of as a sequence of multiple copies of the same neural network through time, where each RNN cell passes on information to its successing cell. Note that the same weight matrices $\boldsymbol{W}_{xh}, \boldsymbol{W}_{hh}, \boldsymbol{W}_{hy}$ are used at every time step, where the weight matrices are used to parametrize input to hidden, hidden to hidden as well as hidden to output connections (Goodfellow et al., 2016). Outputs $\hat{\boldsymbol{y}}_0, \ldots, \hat{\boldsymbol{y}}_t$ can be obtained and loss functions can be computed for every time step. Finally, the total cost for the RNN is obtained by summing up the losses at every time step. Black arrows denote a forward pass. The red arrows illustrate the backward passes of the backpropagation algorithm both to every time step and then from time step t to the beginning of the sequence.

The RNN cell updates its hidden state $\boldsymbol{h}_t$ by taking the sum of the input vector $\boldsymbol{x}_t$ and the previous hidden state vector $\boldsymbol{h}_{t-1}$ multiplied by their respective weight matrices and applying a hyperbolic tanget function to this sum:

$$\boldsymbol{h}_t = tanh(\boldsymbol{W}_{hh}^T \boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}^T \boldsymbol{x}_t). \tag{3.98}$$

The output vector is obtained as a transformed version of the internal state of the RNN cell by multiplying it with a separate weight matrix:

$$\hat{\boldsymbol{y}}_t = \boldsymbol{W}_{hh}^T \boldsymbol{h}_t. \tag{3.99}$$

Note that the $\tanh(\cdot)$ function ensures that the outputs of the values propagated through the network stay in the codomain of $[-1, 1]$, which is important in order to avoid that some values become very large while other values become very small over time in the course of the recurrent computations.

Pascanu, Gulcehre, Cho, and Bengio (2013) consider **deep RNNs** and argue that building a deep recurrent neural network by stacking multiple recurrent hidden states on top of each other can enable the hidden state at each level to operate at different time scales. Moreover, they find that the deep RNN outperformes the basic RNN in the context of language modeling. Hermans and Schrauwen (2013) also argue that the benefit of deep RNN architectures using stacked hidden-to-hidden transitions is the introduced capability to process a times series at different time scales. They also state that the purpose of deep RNNs is to introduce memory instead of hierarchical processing capabilities of hidden layers, which is the main benefit of building deep feed-forward neural networks.

3.4.4 Long Short Term Memory Neural Network (LSTM)

When it comes to time series forecasting, a specific network architecture lends itself well to that purpose, which is that of a **Long Short Term Memory neural network (LSTM)**. The LSTM was first introduced by Hochreiter and Schmidhuber (1997).
According to Brownlee (2017), the LSTM architecture was developed in order to overcome the specific shortcoming of the existing Recurrent Neural Network (RNN) that long time lags were not accessible to this existing architecture and that it was hard to train and scale effectively. RNNs were facing either a "vanishing gradient" or an "exploding gradient" problem, which means that the weight updating procedure either led to very small weight changes that had no effects or to very large changes. Thus, the weight matrices were unstable in either one of the two cases.
A regular LSTM cell possesses two state components in contrast to the regular RNN cell, which are the hidden state $\boldsymbol{h}_t$ that represents the short-term memory component and the internal cell state $\boldsymbol{C}_t$ that provides the function of a long-term memory component (Hewamalage et al., 2019). The LSTM cell also features three gates which control the flow of information through time.

The LSTM cell architecture is shown in Figure 3.9. States in a LSTM cell are updated in the fashion proposed by Hewamalage et al. (2019):

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i * \boldsymbol{h}_{t-1} + \boldsymbol{V}_i * \boldsymbol{x}_t + \boldsymbol{b}_i) \tag{3.100}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o * \boldsymbol{h}_{t-1} + \boldsymbol{V}_o * \boldsymbol{x}_t + \boldsymbol{b}_o) \tag{3.101}$$

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f * \boldsymbol{h}_{t-1} + \boldsymbol{V}_f * \boldsymbol{x}_t + \boldsymbol{b}_f) \tag{3.102}$$

$$\tilde{\boldsymbol{C}}_t = \tanh(\boldsymbol{W}_c * \boldsymbol{h}_{t-1} + \boldsymbol{V}_c * \boldsymbol{x}_t + \boldsymbol{b}_c) \tag{3.103}$$

$$\boldsymbol{C}_t = \boldsymbol{i}_t \odot \tilde{\boldsymbol{C}}_t + \boldsymbol{f}_t \odot \boldsymbol{C}_{t-1} \tag{3.104}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t * \tanh(\boldsymbol{C}_t) \tag{3.105}$$

$$\boldsymbol{z}_t = \boldsymbol{h}_t. \tag{3.106}$$

In Equations (3.100) to (3.106), $\boldsymbol{i}_t$, $\boldsymbol{o}_t$ and $\boldsymbol{f}_t \in \mathbb{R}^d$ are the input gate, output gate and forget gate vectors. The candidate cell state $\tilde{\boldsymbol{C}}_t$, the cell state $\boldsymbol{C}_t$ and the hidden state $\boldsymbol{h}_t$ are also vectors in $\mathbb{R}^d$. Note that the candidate cell state contains the important information that is supposed to be passed on to the future. $\boldsymbol{x}_t \in \mathbb{R}^m$ is the input of the cell and $\boldsymbol{z}_t \in \mathbb{R}^d$ is the output of the cell at time step t. $\boldsymbol{b}_i, \boldsymbol{b}_o, \boldsymbol{b}_f$ and $\boldsymbol{b}_h \in \mathbb{R}^d$ are bias vectors for the input gate, the output gate, the forget gate and the hidden state, respectively. $\boldsymbol{W}_i, \boldsymbol{W}_o, \boldsymbol{W}_f$ and $\boldsymbol{W}_h \in \mathbb{R}^{d \times d}$ are weight matrices of the input gate, output gate, the forget gate and the hidden state vector, respectively. $\boldsymbol{V}_i, \boldsymbol{V}_o, \boldsymbol{V}_f$ and $\boldsymbol{V}_c \in \mathbb{R}^{d \times d}$ denote the weight matrices of the current input. $\odot$ refers to element-wise multiplication which is defined as $(\boldsymbol{A}_{ij} \odot \boldsymbol{B}_{ij}) := (\boldsymbol{A})_{ij}(\boldsymbol{B})_{ij}$. This multiplication is also called the Hadamar product and it is defined for matrices of the same dimensions. The $tanh(\cdot)$ activation function in Equation (3.103) of the candidate cell state outputs values in the codomain of $[-1, 1]$ in order to control the range of the output values of the network just as in the case of the RNN. The hyperbolic tangent function is defined in Equation (3.90).

The forget gate and the input gate in Equation (3.104) determine jointly which information from past time steps to retain and which parts of the current period's information to propagate to the future time steps. The first step in the functioning principles of an LSTM cell is to forget irrelevant parts of information from the previous cell state $\boldsymbol{C}_{t-1}$ by looking at the previous hidden state $\boldsymbol{h}_{t-1}$ and the current input $\boldsymbol{x}_t$, which is done by means of the forget gate through a sigmoid layer. The candidate cell state $\tilde{\boldsymbol{C}}_t$ contains a vector of candidate values created by applying the $tanh(\cdot)$ function to the the previous hidden state and the current input. Next, the input gate selects the amount of relevant new information from the candidate cell state $\tilde{\boldsymbol{C}}_t$ to be added to the cell state $\boldsymbol{C}_t$ by means of a sigmoid layer. Then, these selected candidate values are added to the prior information contained in the previous cell state C_{t-1} through pointwise addition. Finally, the output gate controls what information is sent to the network in the next time step by generating $\boldsymbol{z}_t$ as the product of the elementwise multiplication of the output gate and the modified cell state $\boldsymbol{C}_t$, after the later has been passed through the $tanh(\cdot)$

function. The modified cell state $\boldsymbol{C}_t$ and the current hidden state $\boldsymbol{h}_t$, which is equivalent to the output, are passed over to the next time step. $\sigma(\cdot)$ denotes the sigmoid activation function used by all three gates, which is given in Equation (3.86). The sigmoid function outputs values in the codomain of $[0, 1]$.
Note that a value of 0 in the forget gate means that all information from the previous cell state is discarded by pointwise multiplication, whereas a value of 1 means that the information from the previous cell state is fully retained and a value in between means that a certain part of the information is retained in the internal state parameter C_t, which is built up along the time steps of the time series.

Thus, LSTMs are suitable for time series forecasting for three reasons: First, they can process input and output sequences time step by time step, which allows for variable input and output lengths. Second, they incorporate a long-term memory state (the cell state C_t), which can be used to factor in long-term temporal dependencies of input sequences. Third, they can overcome the vanishing and exploding gradient problems characteristic of RNNs by means of their computational block or cell architecture which allows for backpropagation through time with uninterrupted gradient flow (Brownlee, 2017).

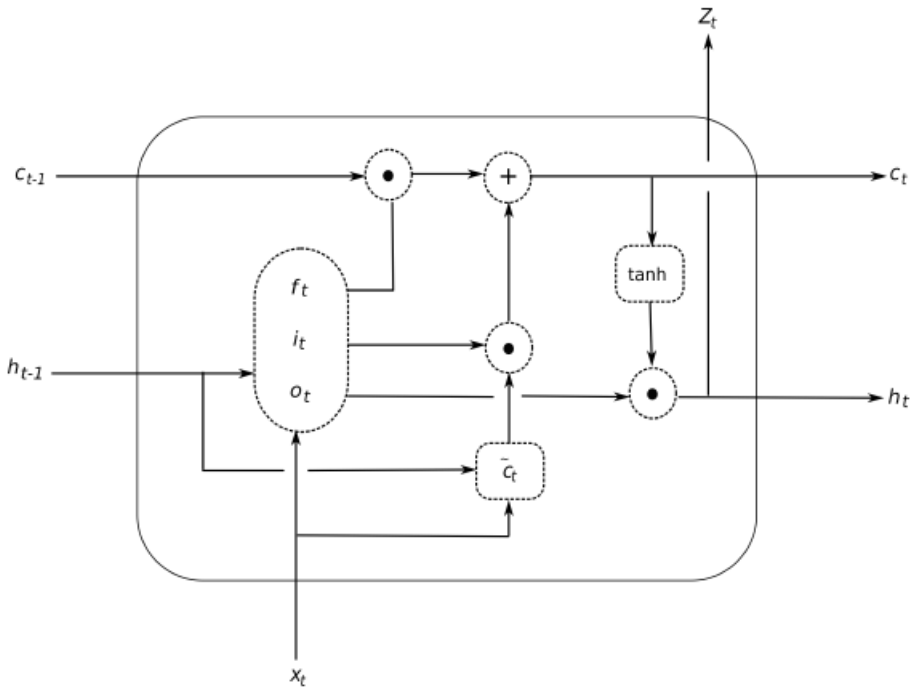

Figure 3.9: LSTM neural network cell architecture

Notes: This figure is taken from Hewamalage et al. (2019) and shows the structure of a regular LSTM cell, where the internal cell state $\boldsymbol{C}_t$ represents the long-term memory component and the hidden state $\boldsymbol{h}_t$ denotes the short-term memory component. The three LSTM gates, i.e. the forget gate, the input gate and the output gate, are denoted by $\boldsymbol{f}_t, \boldsymbol{i}_t$ and $\boldsymbol{o}_t$. These gates control the flow of information through the network across time, ensure the stability of the computational block and overcome the vanishing and exploding gradient problems. $\odot$ denotes element-wise multiplication. $\oplus$ denotes elementwise addition. Note that every gate consists of a hidden layer with a sigmoid activation function to transform the propagated input to values in the interval $[0, 1]$ and elementwise-multiplication in order to gate, i.e. control the flow of information. $\boldsymbol{x}_t, \boldsymbol{z}_t$ denote the input and output of the LSTM cell respectively. $\tanh(\cdot)$ denotes the hyperbolic tangent function.

3.4.5 Gated Recurrent Unit Neural Network (GRU)

Another neural network architecture that seems to lend itself well for the purpose of time series forecasting is that of a **Gated Recurrent Unit neural network (GRU)**, which was first proposed by Cho et al. (2014). Just like a RNN cell, the GRU cell possesses only one component to the cell state, which is the hidden state $\boldsymbol{h}_t$. The cell state can be understood as the memory of the network.

Liu, Wu, and Wang (2018) argue that the Gated Recurrent Unit neural network can learn both short-term and long-term dependencies from the training data and Chung, Gulcehre, Cho, and Bengio (2014) state that a GRU network can also overcome the vanishing gradient problem faced when the network is trained through the Backpropagation through time (BPTT)

algorithm just like the LSTM can.
However, the GRU cell, which is illustrated in Figure 3.10, differs from the LSTM cell shown in Figure 3.9 by the fact that it only uses an update gate and a reset gate, where the GRU update gate has the same function as the forget and the input gate of the LSTM cell combined. Thus, Hewamalage et al. (2019) argue that the GRU network is less complex than the LSTM and faster in computations. This is because there are fewer hyperparameters to train.

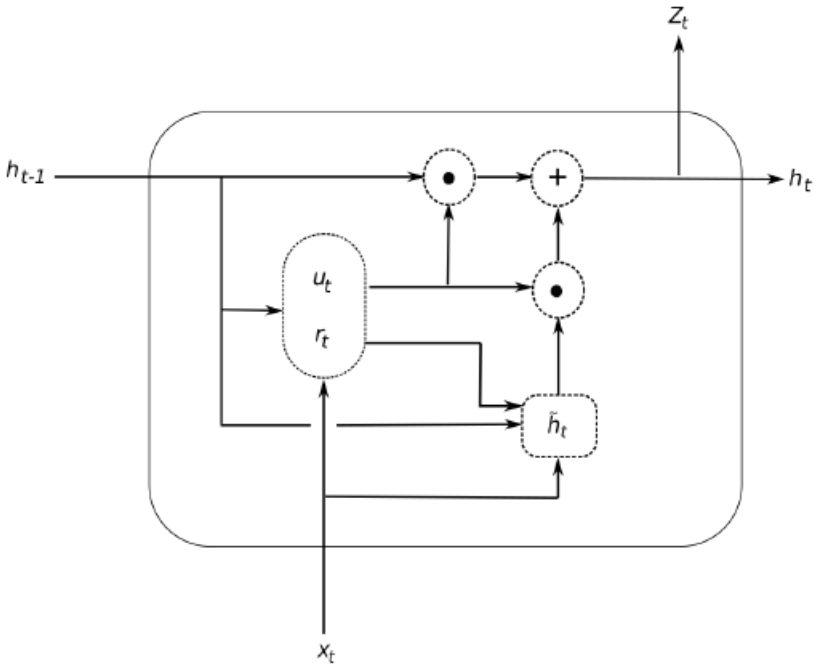

Figure 3.10: GRU neural network cell architecture

Notes: This figure taken from Hewamalage et al. (2019) shows the structure of a GRU neural network cell, where $\boldsymbol{u}_t$ denotes the update gate and $\boldsymbol{r}_t$ denotes the reset gate. The update gate controls how much information to retain by deciding whether to update the current hidden state $\boldsymbol{h}_t$ with information from the candidate hidden state $\tilde{\boldsymbol{h}}_t$. Note that the candidate hidden state may contain information from the hidden state of the previous time step, $\boldsymbol{h}_{t-1}$, and the current input $\boldsymbol{x}_t$ or from the current input only depending on the value of the reset gate. $\boldsymbol{z}_t$ represents the output of the cell at time step t. $\odot$ refers to element-wise multiplication, which is also known as the Hadamard product. $\oplus$ denotes element-wise addition just as in the case of the LSTM.

The states in a GRU neural network cell are updated as follows adhering to the notation suggested by Hewamalage et al. (2019):

$$\boldsymbol{u}_t = \sigma(\boldsymbol{W}_u * \boldsymbol{h}_{t-1} + \boldsymbol{V}_u * \boldsymbol{x}_t + \boldsymbol{b}_u) \tag{3.107}$$

$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_r * \boldsymbol{h}_{t-1} + \boldsymbol{V}_r * \boldsymbol{x}_t + \boldsymbol{b}_r) \tag{3.108}$$

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_h * \boldsymbol{r}_t * \boldsymbol{h}_{t-1} + \boldsymbol{V}_h * \boldsymbol{x}_t + \boldsymbol{b}_h) \tag{3.109}$$

$$\boldsymbol{h}_t = \boldsymbol{u}_t \odot \tilde{\boldsymbol{h}}_t + (1 - \boldsymbol{u}_t) \odot \boldsymbol{h}_{t-1} \tag{3.110}$$

$$\boldsymbol{z}_t = \boldsymbol{h}_t. \tag{3.111}$$

In Equations (3.107) to (3.111), $\boldsymbol{u}_t$, $\boldsymbol{r}_t \in \mathbb{R}^d$ are the update gate and reset gate vectors. The candidate hidden state $\tilde{\boldsymbol{h}}_t$ and the hidden state $\boldsymbol{h}_t$ are also vectors in $\mathbb{R}^d$. $\boldsymbol{x}_t \in \mathbb{R}^m$ is the input of the cell and $\boldsymbol{z}_t \in \mathbb{R}^d$ is the output of the cell at time step t. $\boldsymbol{b}_u, \boldsymbol{b}_r$ and $\boldsymbol{b}_h \in \mathbb{R}^d$ are bias vectors for the update gate, the reset gate and the hidden state, respectively. $\boldsymbol{W}_u, \boldsymbol{W}_r$ and $\boldsymbol{W}_h \in \mathbb{R}^{d \times d}$ are weight matrices of the update gate, reset gate and the hidden state vector, respectively. $\boldsymbol{V}_u, \boldsymbol{V}_r$ and $\boldsymbol{V}_h \in \mathbb{R}^{d \times d}$ denote the weight matrices of the current input. $\sigma(\cdot)$ denotes the sigmoid activation function used by the two gates just as in the case of the LSTM. $\odot$ refers to element-wise multiplication as mentioned in the context of the LSTM.

The update gate in the GRU cell controls how much information from the previous hidden state $\boldsymbol{h}_{t-1}$ to carry over to the current hidden hidden state $\boldsymbol{h}_t$, which helps the GRU network to retain long-term information. The reset gate determines how to combine the new input with the previous cell state by deciding whether the previous hidden state is ignored or not, which is the case if the value of the reset gate is close to 0 (Cho et al., 2014). In the case that the previous hidden state is ignored, the GRU cell resets with the current input only, which is how the GRU cell can drop information considered irrelevant for the future.

3.4.6 Learning Algorithm

Besides the bias terms, the weights $\boldsymbol{W}$ are also learnable parameters inside a neural network, that define the network's forecasting function. Both are estimated or "learned" using an algorithm which minimizes a cost function over a set of training data. A loss function is commonly chosen beforehand based on domain knowledge of the characteristics of the data and on the desired type of prediction, e.g. a classification of the data into three classes to predict a trend or the prediction of one or multiple future values. Any arbitrary function that is smooth and continuous can be used (Bishop, 2006). It is not a requirement that the loss function is the same as the evaluation metric, as long as the former is a proxy close enough to the latter.
Common choices for regression problems, i.e. problems which have a continuous or approximately continuous output, are the MSE and the MAE. The MSE will generate the conditional mean of the forecast, whereas MAE will generate the conditional median of the forecast (Good-

fellow et al., 2016). Throughout this thesis the loss function is the MSE, unless otherwise specified.

Given a specific loss function $L(\hat{y}_i, y_i)$, the loss can be computed based on a labeled data-set of inputs and target outputs. If we consider a training set of n observations, the i-th observation is given by the pair $(\boldsymbol{x}_i, y_i)$. In a time series setting, $\boldsymbol{x}_i$ is a window of lagged variables and y_i is the true single data point immediately following. For a multi-output target, y_i would be replaced by $\boldsymbol{y}_i \in \mathbb{R}^h$. The network function can then be used to generate a forecast of the i-th observation which is given by $\hat{f}(\boldsymbol{x}_i; \boldsymbol{W}) = \hat{y}_i$. The total cost across the entire training data set can then be computed as the sum of the losses for each observation:

$$J(\boldsymbol{W}) = \frac{1}{N} \sum_{i=1}^{n} L(\hat{f}(\boldsymbol{x}_i; \boldsymbol{W}), y_i)), \tag{3.112}$$

where the loss is viewed as a function of the weights $\boldsymbol{W}$, keeping the observations $(\boldsymbol{x}_i, y_i)$ in the training data fixed.

The problem of learning the optimal approximation function $\hat{f}(\cdot)$ can then be stated as a minimization problem with the goal to find the set of weights $\boldsymbol{W}^*$ that minimizes the loss of the data:

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} J(\boldsymbol{W}). \tag{3.113}$$

There is no analytical solution to this optimization problem and iterative techniques have to be used to find the optimal set $\{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, ..., \boldsymbol{W}^{(D-1)}\}$. The numeric procedure used for this purpose is referred to as the **learning algorithm**.

According to Goodfellow et al. (2016), the **gradient descent algorithm** is centered around the property that the function $J(\boldsymbol{W})$, which is minimized according to Equation (3.113), can be decreased by moving into the direction of the negative gradient. A multi-variable function's gradient is defined as the vector of all partial derivatives with respect to each of that function's arguments. For the loss defined in Equation (3.112) the gradient can be formulated as

$$\nabla J(\boldsymbol{W}) = \frac{\partial J(\boldsymbol{W})}{\partial w_i} \quad , \forall\ i \in \{1, 2, ..., |\boldsymbol{W}|\}. \tag{3.114}$$

The gradient describes what variables' changes the loss function is most sensitive to around the point of $\boldsymbol{W}$. The direction in the parameter space that results in the steepest ascent of the loss function is therefore given by the gradient. Analogously, the negative gradient describes in what direction the function will decrease by the most. This information can thus be used to take small steps in the direction of the loss function's minimum.

According to Amini and Soleimany (2020a), the gradient descent algorithm involves the following steps: First, the weights are randomly initialized at $\boldsymbol{W_0}$. Then, the gradient given in Equation (3.114) is computed and the weights of the τ-th iteration are updated as follows:

$$\boldsymbol{W}_\tau = \boldsymbol{W}_{\tau-1} - \eta \nabla J(\boldsymbol{W}_{\tau-1}), \tag{3.115}$$

where $\eta > 0$ is the **learning rate** which is a scalar number that determines the size of the step which is taken in the direction of the negative gradient at each iteration. According to Amini and Soleimany (2020a), setting the learning rate can have a huge impact on the performance of the neural network. They argue that the gradient descent algorithm can get stuck in one of the local minima if the learning rate is set too small. However, if η is set too high, then the case can occur that the algorithm diverges and therefore never reaches a minimum.
Next, a small step is undertaken in the direction of the negative gradient and then the entire process is repeated until convergence to a local minimum. At the local minimum there is no longer a direction in which the function can decrease, so we have $\nabla J(\boldsymbol{W}) = \boldsymbol{0}$. This is also a necessary condition for an extremum.

Figure 3.11 illustrates the path of convergence to a local minimum of the gradient descent algorithm when considering the case of a exemplary cost function $J(w_0, w_1)$ based on a set of weights $\boldsymbol{W} = \{w_0, w_1\}$ which contains two scalar weights only.
Note that the learning rate η can also be set adaptively instead of setting it equal to a fixed scalar value. There are several gradient descent algorithms and stochastic gradient descent is just one of them.

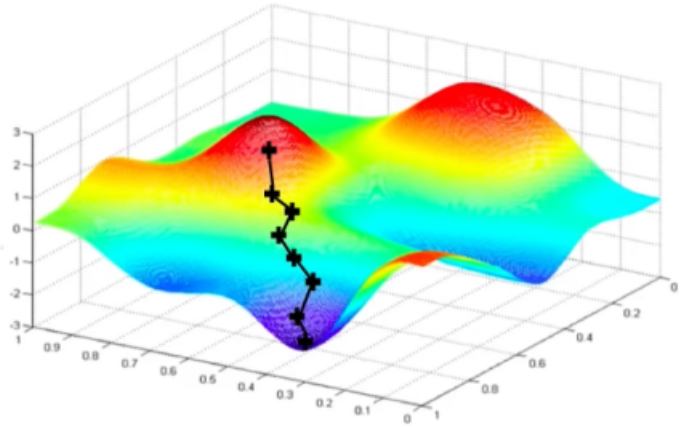

Figure 3.11: Loss optimization using the gradient descent algorithm

Notes: This figure taken from Amini and Soleimany (2020a) shows a three dimensional plot of the the parameter space, where weight w_1 is depicted on the x-axis (northeastern direction), weight w_0 is shown on the y-axis (northwestern direction) and the cost function $J(w_0, w_1)$ is shown on the z-axis (northern direction). This parameter space represents the set of possible cost function values depending on the value of $\boldsymbol{W}$, which is optimized according to Equation (3.113). It can also be seen in this figure how the gradient descent algorithm converges to a local minimum, which happens to be a global minimum in this illustration. First, the weights are initialized at a random value of (w_0, w_1), then the gradient is computed with respect to the weights to find the direction of the steepest ascent. After that, a step of a size equal to the learning rate η is undertaken in the opposite direction, the weights are updated and this process is repeated until a local minimum is reached.

As shown by Choromanska, Henaff, Mathieu, Arous, and LeCun (2015), the cost function of neural networks is highly non-convex by nature, which is caused by the hidden layers in the architecture. This can potentially lead to a case in which the gradient descent procedure is not able to find a global minimum but a local minimum or even a saddle point instead. In practice, however, Choromanska et al. (2015) state that as the size of the network increases most local minima are equivalent and yield similar performance on a test set and the probability of finding a "bad" local minimum quickly decreases.

Goodfellow et al. (2016) state that the gradient descent method, which follows the gradient across the entire training set downhill, can become computationally too extensive when the size of the training set grows large since the time to take a single gradient step becomes prohibitively long. For this reason, they also argue that an extension of the gradient descent method, the **stochastic gradient descent**, can be employed to accelerate the process through taking the average gradient of a minibatch of m independently and identically distributed random samples from the data-generating distribution. The proposition of the stochastic gradient descent extension is therefore to obtain an unbiased estimate of the gradient which allows to follow the gradient of randomly selected minibatches of the training set downhill.

Ruder (2017) provides a comprehensive overview of common gradient descent optimization algorithms. Further optimization algorithms that build on regular gradient descent include the Adagrad and the Adadelta algorithms. According to Ruder (2017), **Adagrad** adapts the learning rate to the parameters and performs larger updates for infrequent and smaller updates for frequent parameters at every time step, whereas regular gradient descent uses the same learning rate for all parameters. While Adagrad eliminates the need to manually tune the learning rate, the problem of a shrinking learning rate that eventually becomes infinitesimally small might occur which prevents the algorithm from obtaining any further knowledge. This problem arises due to the storage of all past gradients. **Adadelta** addresses this problem by restricting the window of accumulated past gradients, which results in a recursively defined decaying average of past squared gradients, and it also does not require to set up a default learning rate.
Another popular learning algorithm, which also uses an adaptive learning rate and which also stores a decaying average of the past squared gradients, is the **Adaptive moments algorithm (ADAM)** proposed by Kingma and Ba (2014). The authors argue that the ADAM algorithm, which is based on adaptive estimates of the first and second moment of the gradients, is computationally efficient and suitable for large scale data sets as well as data exhibiting sparse gradients.

Moreover, a computationally efficient method to evaluate the partial derivatives of the cost function $J(\boldsymbol{W})$ with respect to the weights of the network has to be employed in order to subsequently make adjustments to the weights, which is commonly achieved by the **backpropagation algorithm** (Bishop, 2006) that draws upon the chain rule. He also states that the backpropagation procedure can be divided into four steps: First, an input vector $\boldsymbol{x} \in \mathbb{R}^n$ is propagated forward through the network such that the activation of hidden neuron z_j is transformed by the activation function $g(\cdot)$ as follows

$$z_j = g(a_j) = g(w_{0,i} + \sum_{j=1}^{n} x_i w_{j,i}). \tag{3.116}$$

In this way, activations can be computed for all hidden neurons and output neurons.
Applying the chain rule to compute the partial derivative with respect to the weights results in

$$\frac{\partial J(\boldsymbol{W})}{\partial w_{j,i}} = \underbrace{\frac{\partial J(\boldsymbol{W})}{\partial a_j}}_{\text{backpropagation error } \delta_j} * \underbrace{\frac{\partial a_j}{\partial w_{j,i}}}_{\text{hidden node output } z_i}. \tag{3.117}$$

At this point, we introduce the **backpropagation error**, which Bishop (2006) defines as

$$\delta_j = \frac{\partial J(\boldsymbol{W})}{\partial a_j}. \tag{3.118}$$

The link between the partial derivative of the cost function and the backpropagation error is then given by

$$\frac{\partial J(\boldsymbol{W})}{\partial w_{j,i}} = \delta_j z_i, \tag{3.119}$$

which sets the derivative with respect to weight $w_{j,i}$ equal to the product of the backpropagation error δ_j at the output end of the weight and the value for z_i for the node at the input end of the weight.
In the second step, the backpropagation errors for all output units are evaluated, which are obtained through

$$\delta_k = \hat{y}_k - y_k, \tag{3.120}$$

i.e. the difference between the computed output value for the k-th output neuron and the true value for the k-th output neuron. $y_k \in \mathbb{R}$.
According to Bishop (2006), the third step then consists of the backpropagation of the δ_k-terms from nodes higher up in the network according to the following backpropagation formula

$$\delta_j = g'(z_j) \sum_k w_{k,j} \delta_k \tag{3.121}$$

in order to obtain the backpropagation error δ_j for each hidden unit in the network.
Finally, the fourth step consists of the evaluation of the derivatives by means of Equation (3.119), which links the backpropagation errors and the partial derivatives with respect to the weights. These partial derivatives then form the components of the gradient, which is used to update the weights according to Equation (3.115).

3.4.7 Forecasting with Neural Networks

Generally, neural networks have the ability to learn a mapping from inputs to outputs in a broad range of situations, and therefore, with proper data preprocessing, can also be used for time series forecasting. The field of time series forecasting comes with its own set of unique problems, all of which have to be addressed before modeling. Although there is extra information to be extracted from past observations, these also introduce more complexity and one has to be cautious of the time dependencies between the variables.

Forecasts constructed by neural networks can either be set up as one-step ahead or multi-step

ahead forecasts.
For a one-step ahead forecast the fitted model learned from the series $\hat{f}(\cdot)$ returns the following scalar output:

$$\hat{y}_{T+1} = \hat{f}(y_T, y_{T-1}, \ldots, y_{T-p+1}), \tag{3.122}$$

where each of the p neurons in the input layer represents a lagged value of the target in the output layer.

According to Yan (2012), multi-step ahead forecasts can either be obtained through the Direct modelling approach or through the Recursive modelling approach, where the direct modelling approach draws on multiple prediction models one of which is employed for each single period of the h-steps ahead. The recursive approach, on the other hand, consists of a sequence of recursive one step ahead predictions using the prior prediction as an input to the subsequent steps until the forecast horizon h is reached.
Yan (2012) also argues that the direct approach is computationally more extensive due to the fact that multiple models have to be trained, but it also avoids prediction error accumulation, a disadvantage of the recursive modelling approach.
BenTaieb, Bontempi, Atiya, and Sorjamaa (2012) review a third multi-step ahead forecast approach, the direct recursive (DiRec) approach, which combines the architectures of the two previously mentioned approaches. These three methods, the Direct, Recursive and the DiRec approach, model the data as a multi-input single-output function.
However, this single output mapping of the data neglects the stochastic dependencies between future values and therefore the **multi-input multi-output (MIMO) approach** has been developed. Hewamalage et al. (2019) state that the MIMO approach produces the forecasts for the whole output window at once rather than producing forecasts for each time step in isolation.
Implementing the MIMO multi-step ahead forecasting approach also results in some changes to the structure of the network. The input layer is the same as for the one-step ahead forecast, but the output layer now consist of h neurons instead of 1. Each neuron represents one time step of the h steps in the multi-step ahead forecast. The neural network thus outputs a vector $\hat{\boldsymbol{y}} \in \mathbb{R}^h$ corresponding to the whole forecasting horizon of h steps ahead. Thus, the following multi-output vector containing the forecasts is returned by the MIMO approach in one computational step :

$$\hat{\boldsymbol{y}}^T = (\hat{y}_{T+h}, \ldots, \hat{y}_{T+1}) = \hat{\boldsymbol{f}}(y_T, y_{T-1}, ..., y_{T-p+1}), \tag{3.123}$$

where $\hat{\boldsymbol{f}}(\cdot)$ is the fitted multi-output model learned form the time series $\{y_t\}_{t=1}^T$ when p lags

of the series are taken as an input for the MIMO approach. Note that $\boldsymbol{f}(\cdot)$ is a vector-valued function that carries out the mapping $\boldsymbol{f} : \mathbb{R}^p \mapsto \mathbb{R}^h$.

BenTaieb et al. (2012) conduct an extensive comparison of the five common approaches to multi-step ahead forecasting and find that the MIMO approach exhibits superior performance compared to the three aforementioned multi-input single-output approaches. They also review a second multi-input multi-output approach, the direct multi-output (DIRMO) approach, which is an attempt to incorporate the tradeoff between preserving stochastic dependencies between forecasted values and preserving the flexibility of the modelling approach. BenTaieb et al. (2012) state that the DIRMO approach involves separating the forecast horizon h into blocks and then forecasting these blocks in MIMO fashion, however, they find that the MIMO strategy exhibits the best performance, i.e. the lowest forecast error, while the DIRMO comes in as a close second best approach. For the purpose of this paper, we therefore employ the MIMO approach for all more than multi-step ahead forecasts produced by neural networks. In doing so, we follow the approach taken by Hewamalage et al. (2019), who also obtain their multistep-ahead forecasts for RNN architecture neural networks through a MIMO approach.

The stochastic dependencies can be preserved by using the **rolling window approach**. The most common way to feed time series data into a neural network is to break the whole time series into consecutive input windows and then get the neural network to predict the single point or window immediately following the input window (2019). The input window referred to simply consists of the p lagged values of the target value to be forecasted at time T, y_{T+1}. Figure 3.12 shows how the rolling window approach can be applied to a sample time series. Bandara et al. (2019) state that the rolling window approach transforms a time series $\{y_t\}_{t=1}^{T}$ in to pairs of $< input, output >$ batches, which can then be used for training a LSTM or a GRU recurrent neural network. For a time series of length T, the series is converted into $(T - p - h)$ patches of size $(p + h)$, where p is the size of the training input window and h is the size of the training output window. The application of the MIMO multi-step forecasting approach requires the application of the rolling window technique for these recurrent neural network architectures. Every recurrent cell then accepts a window of inputs, i.e. an array of lagged values, and produces a window of outputs succeeding the last input time step. Subsequently, the windows are shifted forward by one step and the process is repeated until the last patch of the training set.

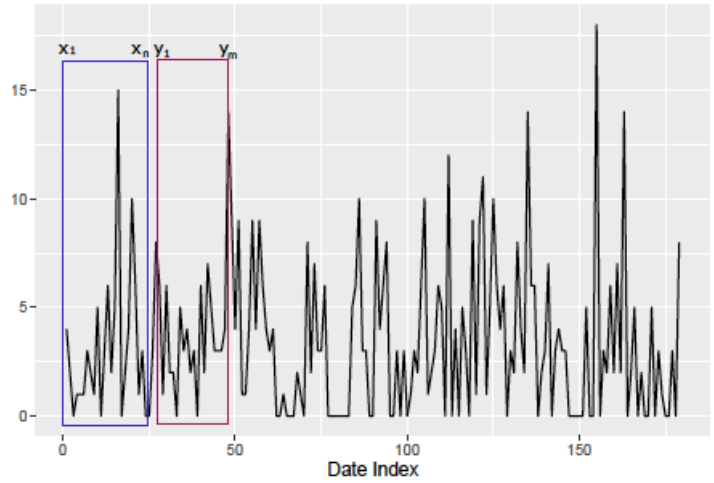

Figure 3.12: Rolling window approach application example

Notes: This figure is taken from Bandara et al. (2019) and shows how the rolling window approach can be applied to an exemplary time series. The input window is given by $\{x_1, x_2, \ldots, x_p\}$ and the output window is given by $\{y_1, y_2, \ldots, y_h\}$, where p and h are the training input window size and the training output window size, respectively. The initial input window (shown in blue) and the initial output window (shown in red) are then shifted forward one time step at a time until the last $< input, output >$ batch is reached. The series is then split up into training set and test set as usual.

Pan and Politis (2016) argue that statistical inference is incomplete without a measure of inherent accuracy. They point out that in case of point estimators, which use observed data to estimate a model parameter, accuracy can either be measured by a standard error or by confidence intervals, whereas in case of point forecasts, accuracy is typically described by the error variance or by a prediction interval.

Khosravi, Nahavandi, Creighton, and Atiya (2010) argue that a fast and reliable way to obtain prediction intervals for neural networks is to set up a separate neural network with two outputs in order to estimate the upper and lower bounds of the prediction intervals by training the network to minimize a prediction interval based cost function. The authors also propose a sophisticated method to obtain PIs, which they call the lower upper bound estimation (LUBE) method. The LUBE method takes the CWC measure, which incorporates both coverage probablity and mean width of the PI, as a loss function for the neural network used to approximate the PI bounds and draws on the simulated annealing method to minimize the CWC. Simulated annealing is deployed since the CWC measure is nonlinear and non-differentiable and thus, stochastic gradient descent cannot be used. However, the risk of the latter method to get trapped in local minima is also avoided in this way.

In this paper, we use a **pinball loss function** to train a separate neural network in order to

estimate the upper and lower bounds of the prediction intervals as Smyl (2020) suggests. A graphical illustration is shown in Figure 3.13. The **pinball loss function**, which is the loss function of a quantile regression, is defined by Romano, Patterson, and Candes (2019) as

$$\rho_\alpha(y, \hat{y}) = \begin{cases} \alpha(y - \hat{y}) & \text{if } y - \hat{y} \geq 0 \\ (1 - \alpha)(\hat{y} - y) & \text{if } y - \hat{y} < 0 \end{cases}, \tag{3.124}$$

where α is a fixed constant $\in [0, 1]$, which specifies the desired significance level, and which the authors refers to as the miscoverage rate. According to Romano et al. (2019), the aim of quantile regression is to estimate a pre-specified quantile of response variable vector $\boldsymbol{y}$ conditional on feature vector $\boldsymbol{x}$. Thus, the **α-th conditional quantile function** is

$$q_\alpha(x) = \inf\{y \in \mathbb{R} : F(y|X = x) \geq \alpha\}, \tag{3.125}$$

where $F(y|X = x)$ is the conditional distribution function of y given $X = x$. The conditional quantile is then estimated by solving the following optimization problem:

$$\hat{\boldsymbol{\theta}} = \min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \rho_\mu(\boldsymbol{y_i}, f(\boldsymbol{x_i}, \boldsymbol{\theta})), \tag{3.126}$$

where $f(\boldsymbol{x_i}, \boldsymbol{\theta}) = \hat{q}_\alpha(x)$ is the quantile regression function. We use neural networks with a pinball loss function to estimate $q_\alpha(x)$, which can be used to compute the upper and lower prediction interval bounds for a pre-specified significance level α.

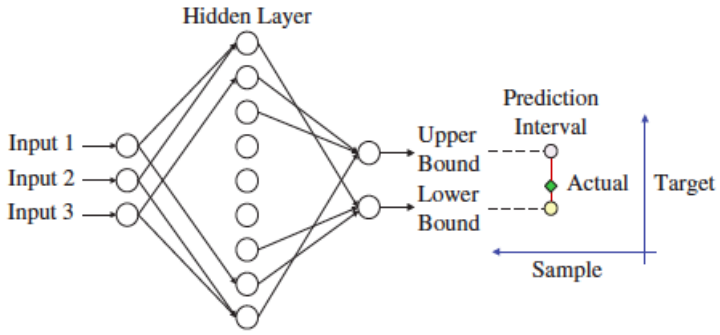

Figure 3.13: PI construction method for neural networks

Notes: This figure is taken from Khosravi et al. (2010) and shows how a symbolic separate neural network model with a PI-based loss function can be employed to estimate the upper and lower bounds of the associated prediction interval. The number of neurons and layers is chosen arbitrarily for exemplary purposes. Note that it is also possible to estimate the upper and lower PI bounds through two separate neural networks with single outputs instead of one single network with two outputs. For a 95% PI, the desired quantile levels would be $\alpha_{low} = 0.025$ and $\alpha_{high} = 0.975$. In this paper, we use a pinball loss function as given in Equation (3.124) and two single output neural networks to estimate each PI bound separately.

Section 4

Data

In this section we describe our source of data, which is the NYC Open Data platform maintained by the Mayor's Office of Data Analytics (MODA) and the Department of Information Technology and Telecommunications (DoITT), and the dimensionality and structure of the data we use for the specific Smart City domains we want to investigate.
NYC Open Data[1] contains machine-readable data sets collected by the city government and supplied by various New York based agencies, including the Metropolitan Transportation Authority (MTA) and the Department of City Planning (DCT), that are made available for public use. We found this platform which has been created as an implementation of a Smart City initiative of the city of New York with the aim to improve the services provided by the city government and the city's various agencies to be particularly useful for our paper.

New York City is the most densely populated city in the entire U.S. with more than 8 million inhabitants living in close proximity. In addition, it is a major hub for multiple industries such as finance, technology and media. The city is thus characterized by a hectic environment which requires sophisticated infrastructure solutions.
This makes New York City a prime candidate for smart city initiatives. In 2015 it vowed to be one of the most technology and data driven cities in the world [2].

4.1 Traffic Flow Data

New York City tracks the data of every single ride taken in a for-hire transportation vehicle operating within city limits. These records account for more than 185,000 drivers and 130,000 vehicles. The main categories are NYC yellow cabs, NYC green cabs and ride-sharing apps

[1] See http://www.nyc.gov/html/data/about.html.
[2] See https://www1.nyc.gov/assets/forward/documents/NYC-Smart-Equitable-City-Final.pdf for the full smart city report of the NYC Mayor's office of Technology and Innovation.

such as Uber, Lyft and Juno. We will only be examining the data for yellow cabs, since they exclusively operate in Manhattan, the most congested area in all of the city. The data is recorded automatically by sensors in each vehicle and is mandated by law unlike in many other major cities. This results in a both unique and massive data set. The data is updated monthly and each row represents a yellow cab ride. Data is available from January 2009 until December 2019.

Preprocessing. The raw data set we use for traffic forecasting consists of the "Yellow Taxi Trip Records" and we use a full year of data starting in July 2018 until June 2019. The raw data consists of 93,337,651 rows and 18 columns. The features contained in the trip records include pick-up and drop-off dates, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types and driver reported passenger counts. In order to be able to use the data in a temporal setting that can be used for forecasting, we group the rides in intervals based on when the trip was started. The variable *tpep_pickup_datetime* is a time stamp in units of hours, minutes and seconds and by varying the size intervals we can vary the frequency. Our preprocessing consists of resampling the series to an hourly frequency and then counting components of the variable *tpep_pickup_datetime*, i.e. the number of pickups by NYC yellow cabs per hour. Thus, we end up with a time series consisting of the number of rides started within every hour within the chosen time frame from July 01, 2018 to June 28, 2019. We remove the last full day (June 30, 2019) from the series because we perceive it as an outlier due to trip reporting overlaps between June an July of 2019. We choose the full day of June 22, 2019 (Saturday) as our test set for a 24 hours ahead forecast. We then use June 23 - June 28, 2019 as a hold out set to do various robustness checks. After preprocessing, we end up with 8,712 rows. Thus, we use the count of started yellow cab rides per hour as a proxy for the total amount of rides per hour in New York City. The variable we will be using as the target variable is discrete in its nature since it is a simple count. This hourly time series has a frequency of 24, the sample size of the training set is set to 8,544, the sample size of the test set is 168. For the main forecasting analyses we use 24 hours of the data from the hold out set and for the robustness checks we use different subsets of the hold out set.

The time plot of the entire time series collected for the traffic flow data and that of the last month of data which includes the test set are shown in Figure 4.1.
Table 4.1 shows the summary statistics for the training set and the test set of the traffic flow time series. It can be seen from this table that there are no missing observations and that the original series is strictly positive.

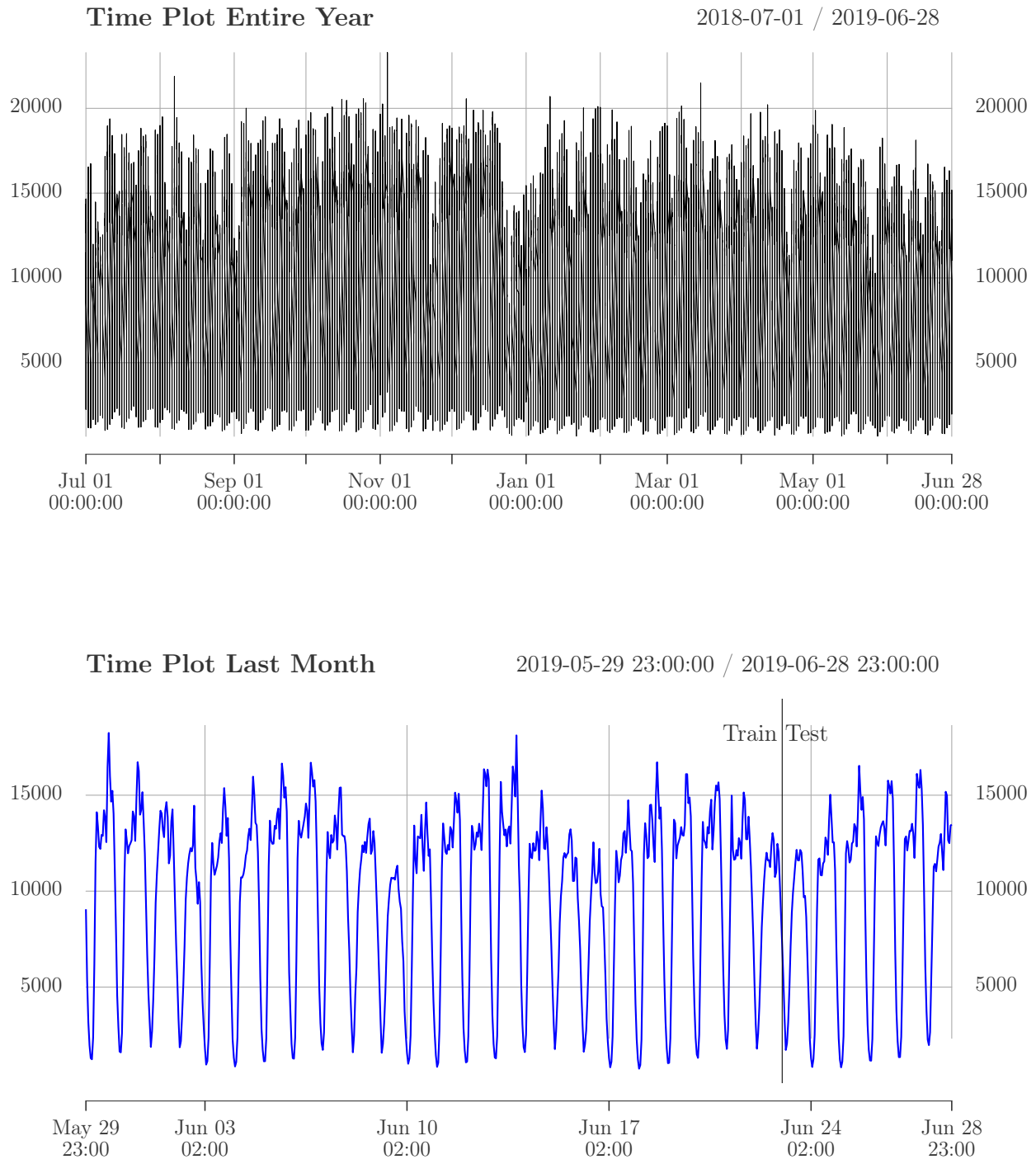

Figure 4.1: Time plots of the hourly traffic flow series

Notes: The top panel shows a time plot of the entire traffic flow time series which contains the number of yellow cab pickups per hour in New York City from July 01, 2018 00:00 to June 28, 2019 23:00. The data set "Yellow Taxi Trip Records" provided by the NYC Taxi & Limousine Commission was obtained from https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. The bottom panel shows the last month of the traffic flow data including the split between train and test set as well as the entire test set.

Table 4.1: Summary statistics for the hourly traffic flow data

Metric	Training Set	Test Set
nobs	8544.00	168.00
NAs	0.00	0.00
Minimum	661.00	818.00
Maximum	23288.00	16532.00
1. Quartile	6432.00	5922.25
3. Quartile	14485.50	12810.50
Mean	10690.03	9683.88
Median	12433.50	11464.50
Sum	91335636.00	1626892.00
SE Mean	55.75	343.97
LCL Mean	10580.74	9004.80
UCL Mean	10799.32	10362.96
Variance	26557448.36	19876607.40
Stdev	5153.39	4458.32
Skewness	-0.53	-0.68
Kurtosis	-0.94	-0.86

Notes: This table contains the summary statistics for traffic flow time series split up by training set and test set as shown in Figure 4.1 as obtained by the *basicStats* function in R. The default confidence interval for the computation of the lower confidence level (LCL) and the upper confidence level (UCL) of the mean is 95%.

4.2 Emergency Medical Services (EMS) Data

The second data set is also obtained from NYC Open Data and contains EMS incident dispatch data generated by the EMS computer aided dispatch system. This data set is provided by the Fire Department of New York City (FDNY) and encompasses emergency incident related data from January 2008 to February 2020. Note that specific locations of the emergency incidents are not reported due to personal identification data protection reasons in accordance with the Health Insurance Portability and Accountability Act. The raw data set comprises 216,556,163 rows and 31 columns. The columns include data on features such as a unique EMS incident identifier, the date and time the incident was created in the dispatch system as well the borough of the incident location. The target variable of interest for this analysis is the number of ambulances dispatched to provide EMS in response to emergency incidents per hour for a specific area of New York City in order to facilitate EMS resource planning.

Preprocessing. We extract the column *First_Assignment_Datetime* which contains time stamps for the first assignment of an EMS vehicle to each specific incident. Next, we subset the data and include only the Bronx borough of New York City in order to provide the EMS demand forecasts for a specific geographical region of the city. We resample the dispatch data for the Bronx borough to an hourly frequency to create a count of ambulance dispatches per hour which are related to emergency incidents as a proxy for EMS demand. We also choose a time frame from January 01, 2019 to December 31, 2019. We end up with a time series which includes 8,760 rows. The sample size of the test set is 24.
We remove the last nine full days of the collected series (December 23, 2019 - December 31, 2019) in order to avoid holiday based distortions. We choose the full day of December 16, 2019 (Monday) as our test set for a 24 hours ahead forecast. We then use December 17 - December 22, 2019 as a hold out set to do various robustness checks. After preprocessing, we end up with 8,544 rows. Thus, we use the count of started yellow cab rides per hour as a proxy for the total amount of rides per hour in New York City. The variable we will be using as the target variable is discrete in its nature since it is a simple count. This hourly time series has a frequency of 24, the sample size of the training set is set to 8,376, the sample size of the test set is 168. For the main forecasting analyses we use 24 hours of the data from the hold out set and for the robustness checks we use different subsets of the hold out set.

The time plot of the entire preprocessed series and that of the last month of data from the EMS series, which includes the test set are shown in Figure 4.2 and the summary statistics are shown in Table 4.2.

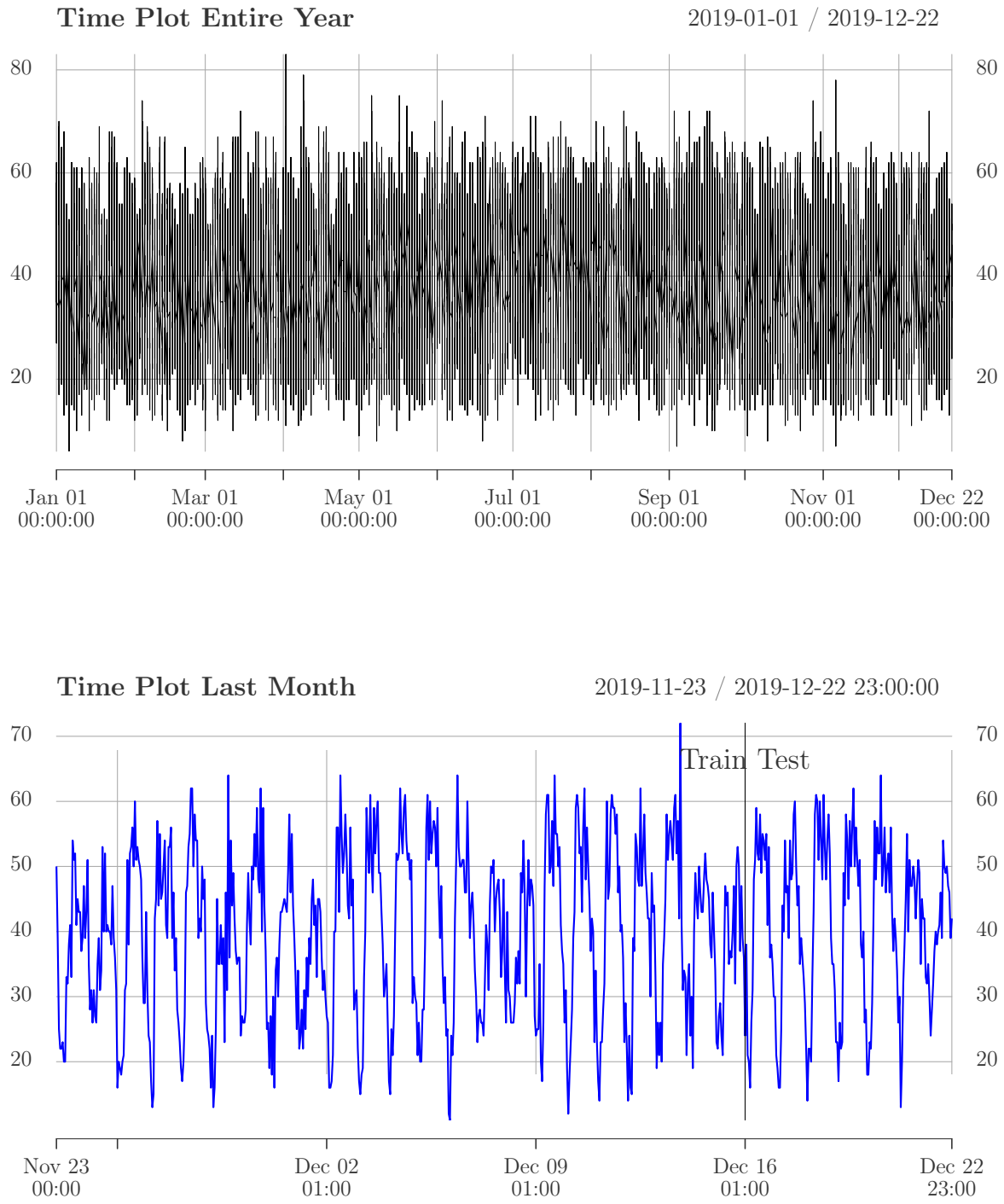

Figure 4.2: Time plots of the hourly EMS series

Notes: The top panel shows a time plot of the entire EMS time series which contains the number of dispatched EMS vehicles per hour for the New York City borough "Bronx" from January 01, 2019 00:00 to December 22, 2019 23:00. The data set "EMS Incident Dispatch Data" provided by the Fire Separtment of New York City was obtained from https://data.cityofnewyork.us/Public-Safety/EMS-Incident-Dispatch-Data/76xm-jjuj. The bottom panel shows the last month of the EMS data including the full test set.

Table 4.2: Summary statistics for the hourly EMS data

Metric	Training Set	Test Set
nobs	8376.00	168.00
NAs	0.00	0.00
Minimum	6.00	13.00
Maximum	83.00	64.00
1. Quartile	29.00	32.00
3. Quartile	51.00	51.00
Mean	40.45	41.18
Median	42.00	43.00
Sum	338824.00	6918.00
SE Mean	0.15	0.97
LCL Mean	40.16	39.27
UCL Mean	40.74	43.08
Variance	179.29	156.65
Stdev	13.39	12.52
Skewness	-0.17	-0.40
Kurtosis	-0.84	-0.84

Notes: This table contains the summary statistics for the EMS time series split up by training set and test set as shown in Figure 4.2 as obtained by the *basicStats* function in R. The default confidence interval for the computation of the lower confidence level (LCL) and the upper confidence level (UCL) of the mean is 95%.

Section 5

Analyses and Results

In this section, the forecasting analyses for the traditional, ensemble and neural network models are presented for both data sets. The choices for model parameters or hyperparameters are explained and the estimation results are reported. Next, the forecasting performance of all models is evaluated with respect to the point forecasts and the prediction intervals by means of the common test error metrics and the common PI metrics.
Subsequently, robustness checks with respect to the data size, the forecast horizon as well as variations of the test set are presented.

5.1 Traffic Forecasting Results from Traditional and Ensemble Models

In order to check whether the series and the seasonally differenced series are stationary or not, an Augmented Dickey Fuller test (ADF) for a unit root is carried out for both series. The results are shown in Table 5.1.
The null hypothesis of the ADF test in the *random walk model* type is $\mathbb{H}_0 : \gamma = 0$, which means that the null of the ADF test is that there is a unit root in the process $\{y_t\}$, i.e. the series is non-stationary. The model $y_t = a_1 y_{t-1} + \epsilon_t$ can be transformed into the equivalent representation $\Delta y_t = \gamma y_{t-1} + \epsilon_t$, where $\gamma = a_1 - 1$. In the other two model specifications, the null hypothesis changes as follows: For the *drift model*, the null is $\mathbb{H}_0 : \gamma = a_0 = 0$ and for the *drift and trend model* the null is $\mathbb{H}_0 : \gamma = a_0 = a_2 = 0$. According to Enders (2014), there are three different regression equations that can be used to estimate γ in order to carry out the Dickey Fuller test for the presence of a unit root in a time series: the pure random walk model, the model with a drift term as represented by an intercept term, and finally the model with a drift and a linear time trend. Those models can then also be expanded by p lagged changes in order to ensure that the ADF auxilliary regression residuals behave like a white noise process,

which is necessary for $\hat{\gamma}$ and $SE(\hat{\gamma})$ to be well estimated.The three following ADF regression equations are equivalent to the three ADF test types "random walk", "drift" and "drift and trend":

$$\Delta y_t = \gamma y_{t-1} + \sum_{i=2}^{p} \beta_i \Delta y_{t-i+1} + \epsilon_t, \tag{5.1}$$

$$\Delta y_t = a_0 + \gamma y_{t-1} + \sum_{i=2}^{p} \beta_i \Delta y_{t-i+1} + \epsilon_t, \tag{5.2}$$

$$\Delta y_t = a_0 + \gamma y_{t-1} + a_2 t + \sum_{i=2}^{p} \beta_i \Delta y_{t-i+1} + \epsilon_t. \tag{5.3}$$

Table 5.1: ADF test results for the time series on traffic volume

	Original series			**Seasonally differenced series**		
	ADF_τ test stat	p-value	lagged changes (p)	ADF_τ test stat	p-value	lagged changes (p)
Test results	-1.291	0.2177	24	-14.7	$< 0.01^{***}$	24
	-1.077	0.2939	48	-13.1	$< 0.01^{***}$	48
	-0.937	0.3441	72	-12.6	$< 0.01^{***}$	72
	-0.604	0.4631	96	-12.8	$< 0.01^{***}$	96
	1% level	5% level	10% level			
Critical values	-2.58	-1.95	-1.62			

Notes: This table contains the test statistics and p-values obtained by carrying out the "random walk" type of the ADF test for a unit root as given in Equation (5.1) for the original as well as for the seasonally differenced series. The number of lagged changes p included in the ADF equations is varied from 24 to 96 in order to account for the autocorrelation at the first four seasonal lags. The level of significance is given by *: p-value < 0.10; **: p-value < 0.05; ***: p-value < 0.01.

Since the collected time series on the traffic flow data does not seem to exhibit a clear trend, we opt for the "random walk" type of the ADF test. The test statistic for the first type of the ADF test is

$$ADF_\tau = \frac{\hat{\gamma}}{SE(\hat{\gamma})}. \tag{5.4}$$

We can see from Table 5.1 that the seasonally differenced series is more suitable for further analysis since the null hypothesis of a unit root can be rejected at the 1% level when autocorrelation at all first four seasonal lags is accounted for, which is not the case for the

original series. According to the conducted ADF tests, the original series can be considered non-stationary since we cannot reject the null hypothesis of a unit root, whereas the one time seasonally differenced series can be considered stationary since we can reject the null hypothesis.

In order to derive the candidate SARIMA model, we use the model selected by the *auto.arima()* R function, which uses the Hyndman-Khandakar algorithm. As described in Section 3.2.1 the algorithm includes unit root testing and minimization of the AIC to select a model. We start our manual modeling procedure by plotting the ACF and PACF of the residuals of the automatically chosen model in order to check whether they resemble a white noise process. Figure 5.1 shows a comparison of the ACF and PACF of the automatically fitted SARIMA $(5,0,5)(2,1,0)_{24}$ model's residuals and those of the finally chosen manual SARIMA $(5,0,5)(4,1,1)_{24}$ model's residuals. After several iterations of residual analysis, we end up with an SARIMA model that has two more seasonal AR terms and one more seasonal MA term. As can be seen in Figure 5.1, the amount of autocorrelation at the lags 48, 72 and 96 is visibly reduced. At lag 24 autocorrelation is only reduced slightly. Increasing the number seasonal parameters is computational infeasible due to instability.

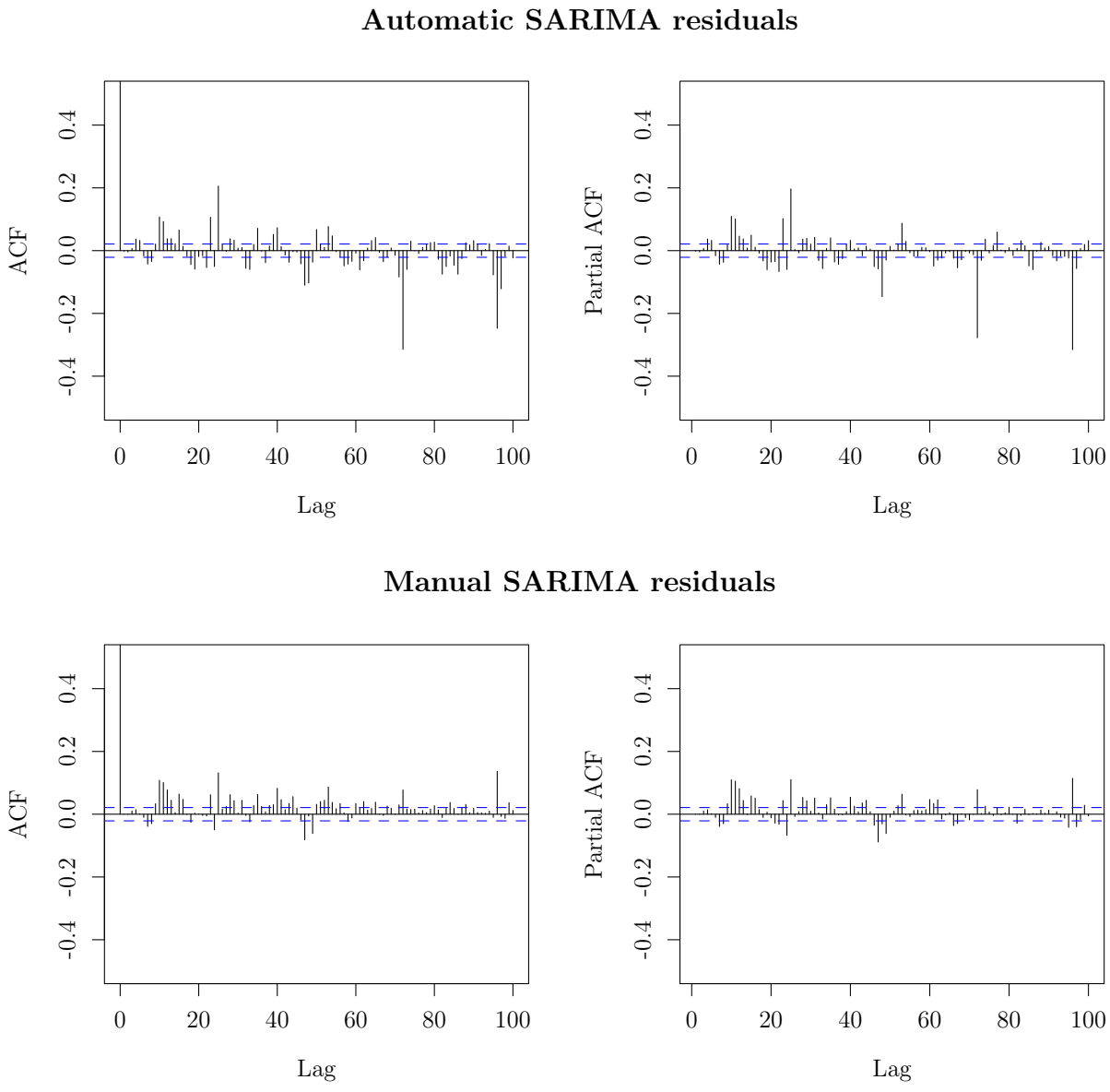

Figure 5.1: ACF/PACF plots comparison of SARIMA model residuals (traffic)

Notes: This figure shows the autocorrelation and the partial autocorrelation for up the 100 lags of the automatically chosen SARIMA $(5,0,5)(2,1,0)_{24}$ model's residuals, the manually refined SARIMA $(5,0,5)(4,1,1)_{24}$ model's residuals and the corresponding confidence bounds at the 95 % level.

Table 5.2 shows the SARIMA model residual diagnostic test results with respect to autocorrelation, normality and conditional heteroskedasticity for both the automatically chosen and the manually refined SARIMA models. We employ a Ljung-Box test for autocorrelation. The test hypotheses of the Ljung-Box test are $\mathbb{H}_0$: *No autocorrelation* and $\mathbb{H}_1$: *autocorrelation* among the model residuals. According to R. J. Hyndman and Athanasopoulos (2018), the test statistic

of the Ljung-Box test is given by

$$Q^* = T(T+2)\sum_{k=1}^{h}(T-k)^{-1}\hat{\rho}_\epsilon(k), \tag{5.5}$$

where $\hat{\rho}_\epsilon(k)$ is the sample autocorrelation in the residuals at lag k and T is the number of observations of the series to be tested. The maximum lag considered when carrying out the test is given by h. We test whether the first h autocorrelations of the model residuals are significantly different from a white noise process, which Enders (2014) defines as a process $\{\epsilon_t\}$ with a mean of 0, a constant variance of σ^2 and with the key property that the elements of the process are serially uncorrelated. The distribution of the test statistic Q^* is a χ^2-distribution with $(h-K)$ degrees of freedom, where K is the number of model parameters.

We also employ a Jarque Bera test for normality with the null hypothesis $\mathbb{H}_0$: *normally distributed residuals* and the alternative hypothesis $\mathbb{H}_1$: *non-normally distributed residuals*. Note that normally distributed data are expected to have a skewness of 0 and a kurtosis of 3. The Jarque Bera test statistic in the form stated by Cromwell, Labys, and Terraza (1994) is given by

$$JB = \frac{T}{6}S + \frac{T}{24}(K-3)^2, \tag{5.6}$$

where S is the sample skewness, K is the sample kurtosis and T is the number of observations in the series. The JB test statistic is $\chi^2(2)$ distributed.

Finally, we use an Engle Lagrange Multiplier test for conditional heteroskedasticity of the model residuals, which is often referred to as "Autoregressive conditonal heteroskedasticity (ARCH) effects". The null hypothesis is $\mathbb{H}_0$: *Squared residuals are a white noise sequence*, which means that the model residuals are homoskedastic. The alternative hypothesis is $\mathbb{H}_1$: *Squared residuals are not a white noise sequence*, i.e. the residuals are heteroskedastic and the model residuals can be described by an ARCH(p) model of the form stated by Enders (2014):

$$\hat{\epsilon}_t^2 = a_0 + a_1\hat{\epsilon}_{t-1}^2 + a_2\hat{\epsilon}_{t-2}^2 + \ldots + a_p\hat{\epsilon}_{t-q}^2 + \nu_t, \tag{5.7}$$

where ν_t is a white noise series. The LM test statistic is TR^2, which is approximately $\chi^2(q)$ distributed, where T stands for the length of the series of model residuals and R^2 is the coefficient of determination from the regression of the squared model residuals on their p lagged values.

Table 5.2: SARIMA model residual diagnostic tests (traffic)

Autocorrelation				
	$\text{SARIMA}(5,0,5)\times(2,1,0)_{24}$		$\text{SARIMA}(5,0,5)\times(4,1,1)_{24}$	
Max lag	**LB test stat(Q^*)**	**P-value**	**LB test stat(Q^*)**	**P-value**
1	0.0235	0.8781	0.0001	0.9902
2	0.1814	0.9133	0.0009	0.9995
3	0.6215	0.8915	0.9141	0.8220
24	492.63	$< 0.0000^{***}$	404.81	0.0000^{***}
48	1295.20	$< 0.0000^{***}$	873.60	0.0000^{***}
72	2465.30	$< 0.0000^{***}$	1154.60	0.0000^{***}
Normality				
	$\text{SARIMA}(5,0,5)\times(2,1,0)_{24}$		$\text{SARIMA}(5,0,5)\times(4,1,1)_{24}$	
	JB test stat	**P-value**	**JB test stat**	**P-value**
	72,271	$< 0.0000^{***}$	175,876	$< 0.0000^{***}$
	Skewness	**Kurtosis** -3	**Skewness**	**Kurtosis** -3
	-0.6060	14.1924	-0.7494	22.1704
Conditional Heteroskedasticity				
	$\text{SARIMA}(5,0,5)\times(2,1,0)_{24}$		$\text{SARIMA}(5,0,5)\times(4,1,1)_{24}$	
Max lag	**LM test stat**	**P-value**	**LM test stat**	**P-value**
4	7,080	$< 0.0000^{***}$	6717	$< 0.0000^{***}$
12	2,329	$< 0.0000^{***}$	1978	$< 0.0000^{***}$
24	953	$< 0.0000^{***}$	443	$< 0.0000^{***}$

Notes: This table shows the test statistics and p-values for the Ljung-Box (LB) test for autocorrelation among the residuals, the Jarque-Bera (JB) test for normality and the Engle Lagrange Multiplier test for conditional heteroskedasticity of the squared model residuals for both the automatically chosen and the manually chosen SARIMA model. The automatically chosen model is an SARIMA $(5,0,5)(2,1,0)_{24}$ model and the manually refined model is an SARIMA $(5,0,5)(4,1,1)_{24}$ model. The level of significance is given by *: p-value < 0.10; **: p-value < 0.05; ***: p-value < 0.01.

Based on the test results for the automatically chosen SARIMA model's residuals shown in Table 5.2, we conclude that we have no serial correlation up to the third regular lag of the residuals but we conclude that serial correlation is present when we vary the maximum lag considered for the test such that the first three seasonal lags are included, which correspond to lags 24, 48 and 72. We also conclude that the residuals are non-normally distributed with a substantial amount of excess kurtosis. From the Engle LM test for conditional heteroskedasticity we conclude that model residuals are heteroskedastic. The conclusions from the residual diagnostic tests are identical for the manually chosen SARIMA model but the excess kurtosis is higher while the skewness is close to that of the automatically chosen residuals. The consequences of these test results are that the assumptions underlying the prediction intervals of the SARIMA models are no longer accurate since the normality assumption is violated. Thus, we calculate the prediction intervals for the SARIMA models using the simulation method of bootstrapping the model residuals to account for the fact that the SARIMA residuals are not normally distributed. We also find evidence for ARCH effects which are not explored further in the course of this paper. Moreover, Best and Wolf (2014) state that the maximum likelihood estimators for the SARIMA coefficients, which are obtained by maximizing a miss-specified Gaussian density function, lose their asymptotic efficiency property but they can still remain consistent and asymptotically normal. They also argue that standard errors for such quasi maximum likelihood estimators are generally too small but asymptotically correct SE estimates can be obtained by using Huber-White standard errors.

The estimated model parameters of the manual SARIMA model are shown in Table 5.3.

Table 5.3: SARIMA $(5,0,5) \times (4,1,1)_{24}$ model estimation results

Dep. Variable	y_T	**No. of Observations**	8544
AICc	139,386.30	**Log Likelihood**	-69,677.10
AIC	139,386.20	**BIC**	139,396.40

Estimated Coefficient		**SE**	**z score**	**P> |z|**
$\hat{\zeta}_1$	0.2900	0.1343	2.1586	0.0309**
$\hat{\zeta}_2$	-0.2239	0.1203	-1.8610	0.0627*
$\hat{\zeta}_3$	0.3032	0.1196	2.5348	0.0113**
$\hat{\zeta}_4$	0.3465	0.1275	2.7164	0.0066***
$\hat{\zeta}_5$	-0.3346	0.0558	-5.9921	< 0.0000***
$\hat{\psi}_1$	0.9530	0.1346	7.0823	< 0.0000***
$\hat{\psi}_2$	1.0481	0.0701	14.9584	< 0.0000***
$\hat{\psi}_3$	0.6283	0.1347	4.6630	< 0.0000***
$\hat{\psi}_4$	0.0083	0.0455	0.1815	0.8560
$\hat{\psi}_5$	0.0056	0.0215	0.2603	0.7946
$\hat{\mathrm{Z}}_1$	0.1941	0.0136	14.3230	0.000***
$\hat{\mathrm{Z}}_2$	-0.2215	0.0119	-18.6429	0.000***
$\hat{\mathrm{Z}}_3$	-0.1551	0.0113	-13.6894	0.000***
$\hat{\mathrm{Z}}_4$	-0.2561	0.0125	-20.4868	0.000***
$\hat{\Psi}_1$	-0.8637	0.0104	-83.2515	0.000***

Notes: The SARIMA model is of the form $(p,d,q) \times (P,D,Q)_m$ and the table shows the estimated regular and seasonal AR and MA coefficients as well as the corresponding p-values when the model is fitted to the training data. The upper and lower bounds for a 95% confidence interval of the estimated coefficients are also given. The Akaike information criterion with small sample correction (AICc) is the AIC value which has been corrected to prevent over-fitting of the regular AIC measure in the case of small samples sizes. The z score is calculated by dividing the estimated coefficient by the standard error (SE). The level of significance is given by *: p-value < 0.10; **: p-value < 0.05; ***: p-value < 0.01.

A graphical illustration of the 24 step ahead forecasts of both the automatically chosen and the manually refined SARIMA model is shown in Figure 5.2.

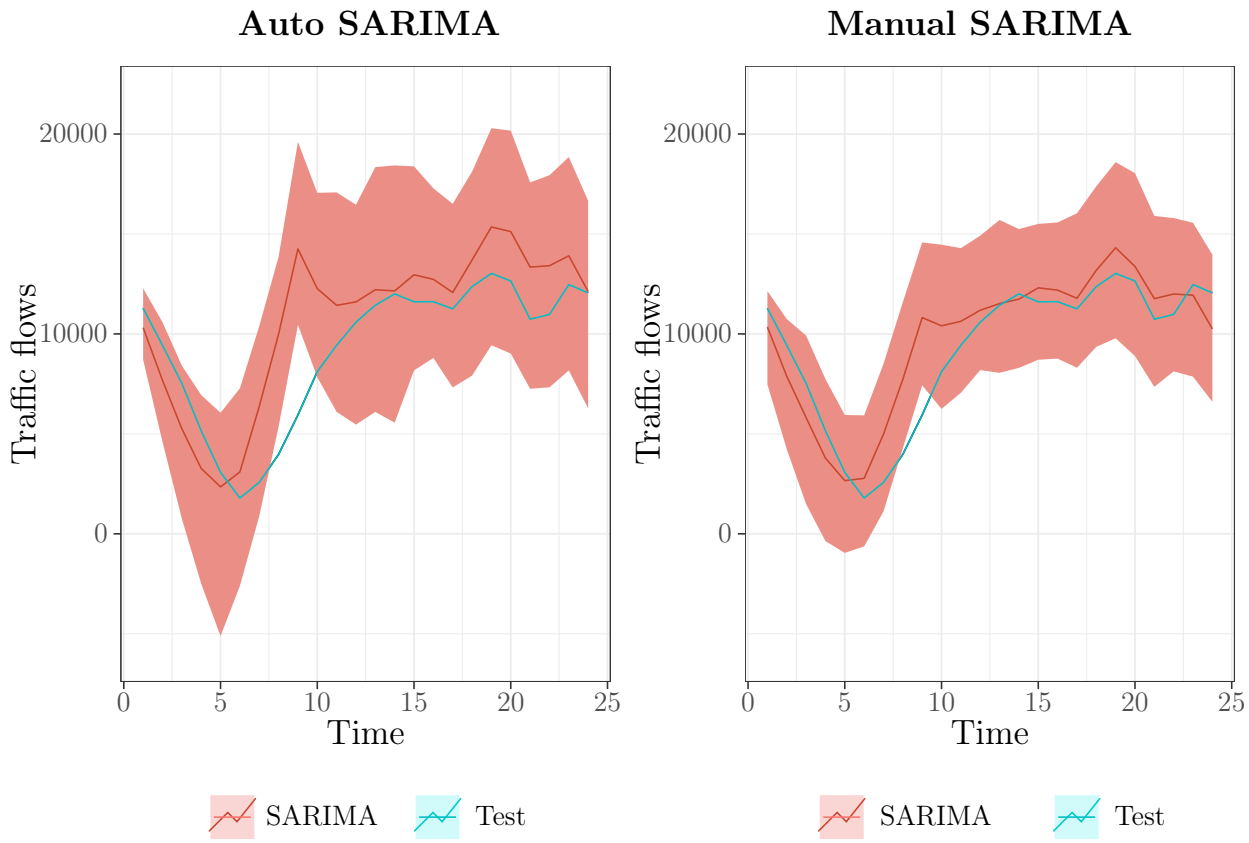

Figure 5.2: Comparison manual vs. automatic SARIMA model (traffic)

Notes: This figure shows the 24 step ahead predictions as well as the bootstrapped 95% prediction intervals of the automatically chosen SARIMA $(5,0,5) \times (2,1,0)_{24}$ model plotted against those from the manually refined SARIMA $(5,0,5) \times (4,1,1)_{24}$ model, both of which are compared against the test set. 100 simulated sample paths are used for bootstrapping the residuals of each model.

Figure 5.2 illustrates that the test set is better approximated by forcasted values of the manual SARIMA model from 12 steps ahead to 24 steps ahead. Also the prediction intervals are narrower compared to the automatic SARIMA model.
Even though the SARIMA model can handle seasonality, there might be multiple patterns of seasonality present in the data. Thus, we will also consider a TBATS model which can be employed to decompose the series into its components including multiple seasonal patterns, if present.

The estimated TBATS model which minimizes the AIC value for the traffic flow data is a TBATS(1, {5,1}, -, <24,8>,<168,5>) model. The *seasonal.periods* parameter was set to $(24, 168)$ in order to model the seasonal period of hourly traffic flows per day and per week respectively. The daily and weakly seasonal periods are evident choices based on a visual

inspection of the time plot of the data.. The *use.arma.errors* option was enabled as well and the *tbats()* function selected a model with an $ARMA(5,1)$ error component, which is included in the modeling of the level of the trend through the α smoothing parameter, which is also the coefficient in the trend level. Since β is not estimated, the $ARMA(5,1)$ errors are not included in the trend growth equation, however. The value of the Box-Cox transformation parameter $\omega = 1$ means that a linear model is chosen and the observations are not Box-Cox transformed. The seasonal period $m_1 = 24$ (daily seasonality) can be described by $k_1 = 8$ harmonics. Next, the second seasonal period $m_2 = 168 = 24 * 7$ (weekly seasonality) can be modeled by $k_2 = 5$ harmonics. **??** shows a trigonometric decomposition of the traffic flow series.

The BATS model is also fitted with a specification of two seasonal periods to model a daily and a weekly seasonal pattern. Table 5.4 contains the smoothing, transformation parameters and the ARMA error coefficients for both models. In contrast to the estimated TBATS model, the BATS model uses a Box-Cox transformation and a dampened trend. Figure 5.3 illustrates the obtained 24 step ahead point forecasts obtained for both models and the respective PIs. We can see that the BATS forecasts approximate the test set better than those of the TBATS across the entire forecast horizon of 24 steps ahead.

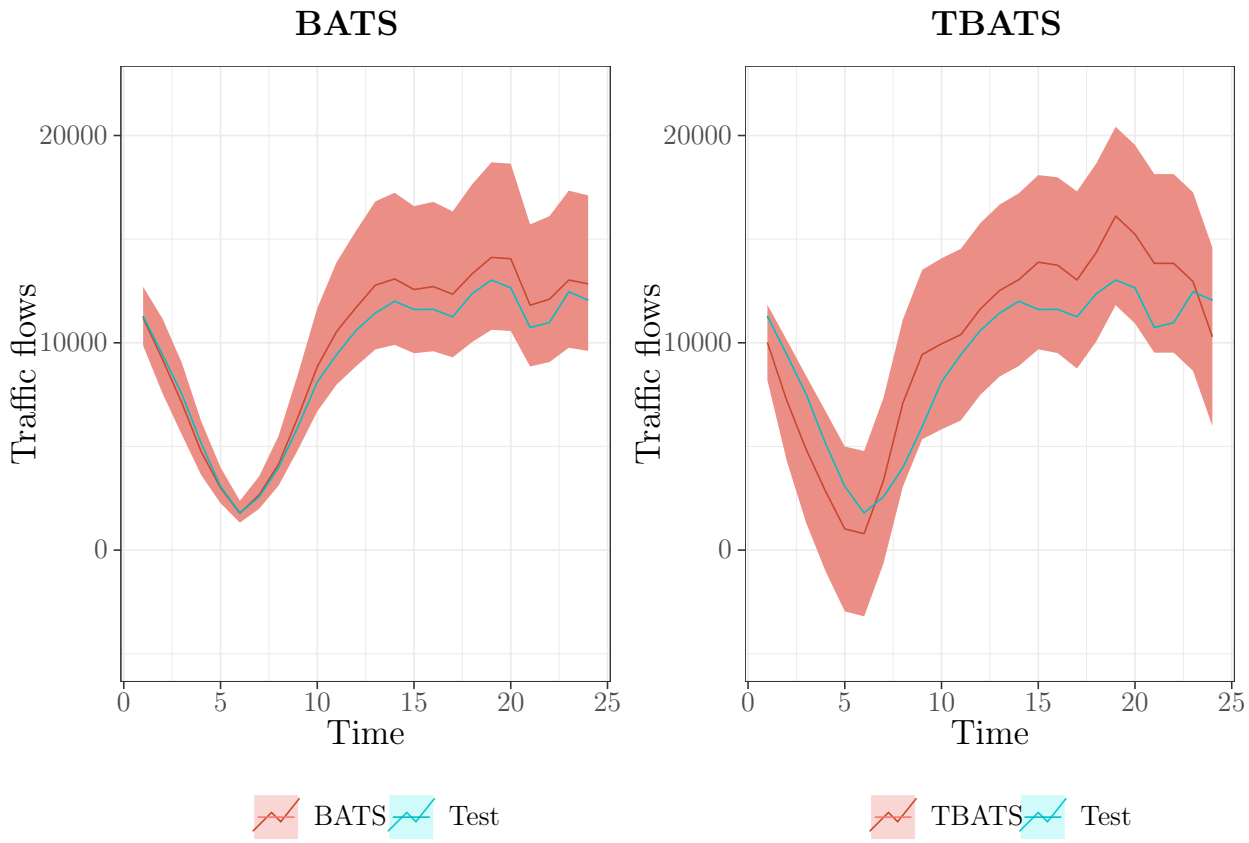

Figure 5.3: Comparison BATS vs. TBATS (traffic)

Notes: This figure shows the 24 step ahead predictions as well as the bootstrapped 95% prediction intervals of the BATS(0.035, {4,1}, 0.934, 24,168) model plotted against those from the TBATS(1, {5,1}, -, <24,8>,<168,5>) model, both of which are compared against the test set. 100 simulated sample paths are used for bootstrapping the residuals of each model.

Table 5.4: BATS and TBATS model estimation results

TBATS model

Transformation and smoothing parameters

$\hat{\omega}$	$\hat{\phi}$	$\hat{\alpha}$	$\hat{\beta}$	$\hat{\gamma}_1^{(1)}$	$\hat{\gamma}_2^{(1)}$	$\hat{\gamma}_1^{(2)}$	$\hat{\gamma}_2^{(2)}$
1.0000	NA	0.0375	NA	$7.03 * 10^{-6}$	0.0002	-0.0004	$7.20 * 10^{-5}$

AR coefficients					**MA coefficients**
$\hat{\zeta}_1$	$\hat{\zeta}_2$	$\hat{\zeta}_3$	$\hat{\zeta}_4$	$\hat{\zeta}_5$	$\hat{\psi}_1$
1.1428	-0.3574	-0.0205	-0.0473	-0.0817	0.0686

AIC

194,232.10

BATS model

Transformation and smoothing parameters

$\hat{\omega}$	$\hat{\phi}$	$\hat{\alpha}$	$\hat{\beta}$	$\hat{\gamma}_1$	$\hat{\gamma}_2$
0.0350	0.9337	0.0530	$2.51 * 10^{-5}$	0.0609	-0.0092

AR coefficients				**MA coefficients**
$\hat{\zeta}_1$	$\hat{\zeta}_2$	$\hat{\zeta}_3$	$\hat{\zeta}_4$	$\hat{\psi}_1$
0.4006	0.5482	-0.3006	-0.1215	0.7005

AIC

186,036.10

Notes: This table shows the estimated smoothing, dampening and transformation parameters as well as the ARMA error process coefficients. ω is the Box-Cox transformation parameter. ϕ is a trend dampening parameter. The smoothing parameter α is the ARMA error term coefficient in the level of the trend and β, the second smoothing parameter, is the ARMA error coefficient in the trend growth equation. $\gamma_1^{(i)}$, $\gamma_2^{(i)}; i = 1, 2$ are seasonal smoothing parameters used for the trigonometric formulation of the two seasonal components in the TBATS model and γ_1 and γ_2 are the seasonal smoothing parameters used in the BATS model. $\zeta_1, \ldots, \zeta_5$ are AR coefficients and ψ_1 is an MA coefficient used to parametrize the respective ARMA errors process selected by both models to account for autocorrelation in the residuals.

To illustrate the ability of the BATS model to handle multiple seasonal patterns, we show a comparison of autocorrelation plots of the model's residuals against those from the manual

SARIMA model, which are shown in Figure 5.4.

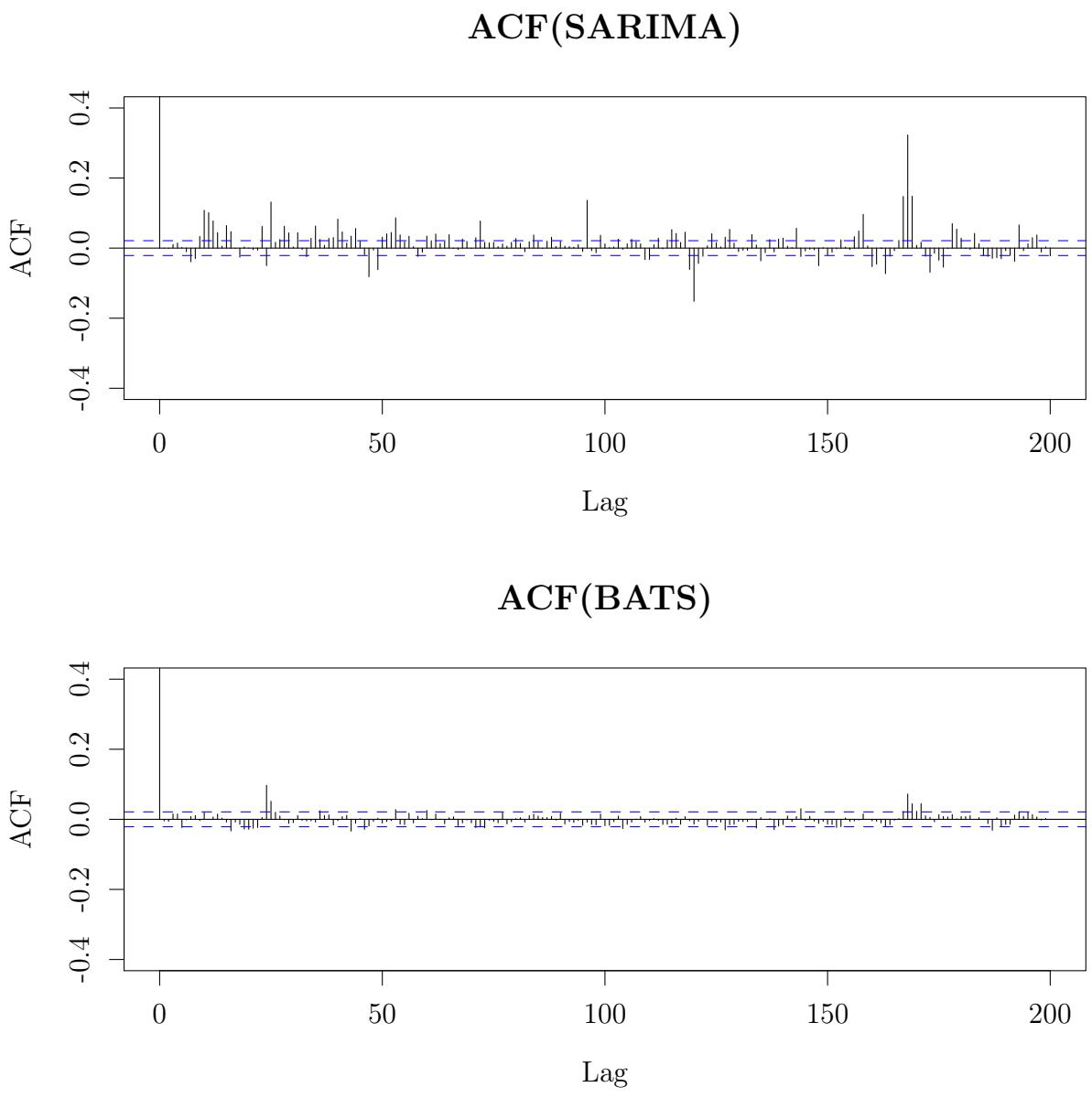

Figure 5.4: Residual analysis BATS vs. SARIMA

Notes: This figure shows ACF for the residuals of both the BATS and the manual SARIMA model for up to 200 lags so the autocorrelation at the weekly level can be compared. The weekly seasonal period is 168.

When a time horizon of 200 lags is considered, it becomes apparent that the autocorrelation in the respective model residuals is clearly reduced over the entire history of lags illustrated in Figure 5.4. Especially the weekly seasonality at lag 168 is captured by the BATS model but not by the SARIMA model.

The next model we employ to forecast the traffic flows is a hybrid model combining the machine learning method of bagging with an exponential smoothing model. In particular, we deploy the model "Bagged.BLD.MBB.ETS", which is the bagged ETS model proposed by Bergmeir et al. (2016) that we explain in more detail in Section 3.3.1. First, the traffic flow time series is Box-Cox transformed, then the series is decomposed into the trend, the seasonal and the remainder components and finally, the remainder is bootstrapped using the moving block bootstrap method and then combined with the trend and the seasonal components to form the bootstrapped series. Figure 5.5 shows the training data plotted against 10 bootstrapped series created in this way to illustrate the bootstrapping technique for time series.

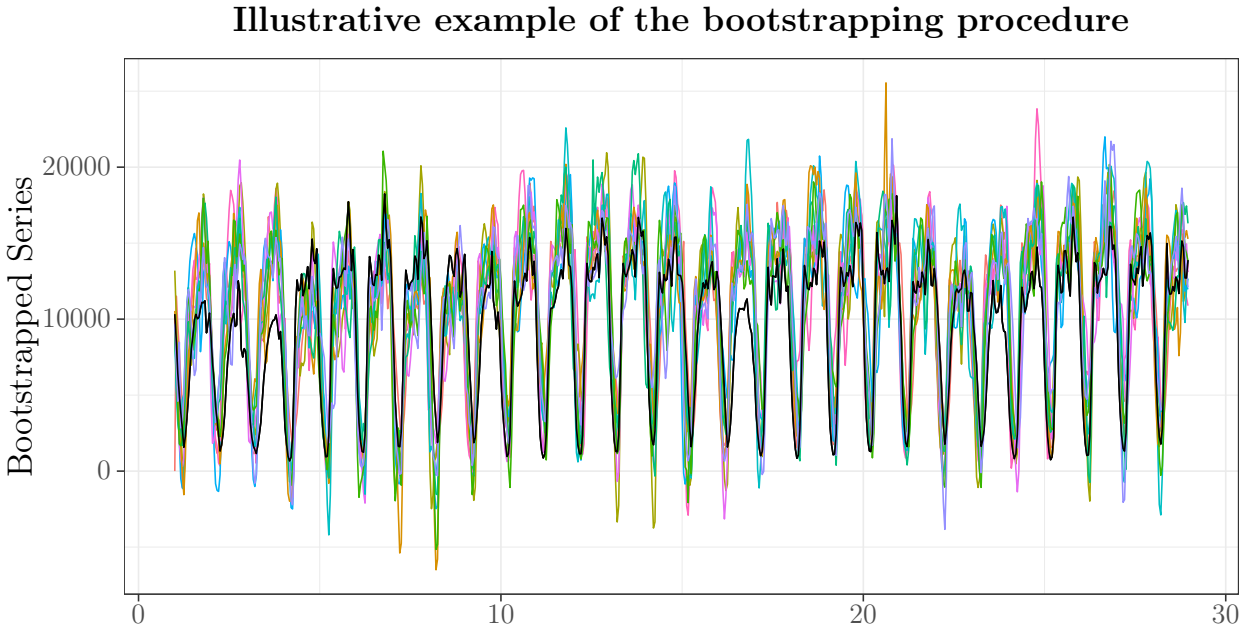

Figure 5.5: Plot of the bootstrapped training data from the traffic flows series

Notes: This figure shows the training data (black) from the traffic flow time series, which are plotted against 10 bootrapped series created through the moving block bootstrap procedure that draws on a Box-Cox transformation and a Loess-based decomposition which we implement by using the R function *bld.mbb.bootstrap()*. In the final ensemble model, 100 such bootstrapped series are generated and the *ETS* function is applied to each of these 100 bootstrapped series.

However, the full baggedETS model consists of an ensemble of 100 ETS models that are fitted to 100 bootstrapped series created in the showcased way. The point forecasts of all 100 ETS models are then combined through the median as suggested by Bergmeir et al. (2016) in order to produce the final point forecasts of the baggedETS model. Prediction intervals are calculated in the following way: First, 100 bootstrapped series are created from the training data using

the MBB technique. Then, an ETS model is fit to each of the series and subsequently refit to the training data. The refitted models are then used to create a simulated distribution of the point forecasts, from which the 0.025 and 0.975 quantiles are taken in order to obtain 95% prediction intervals.

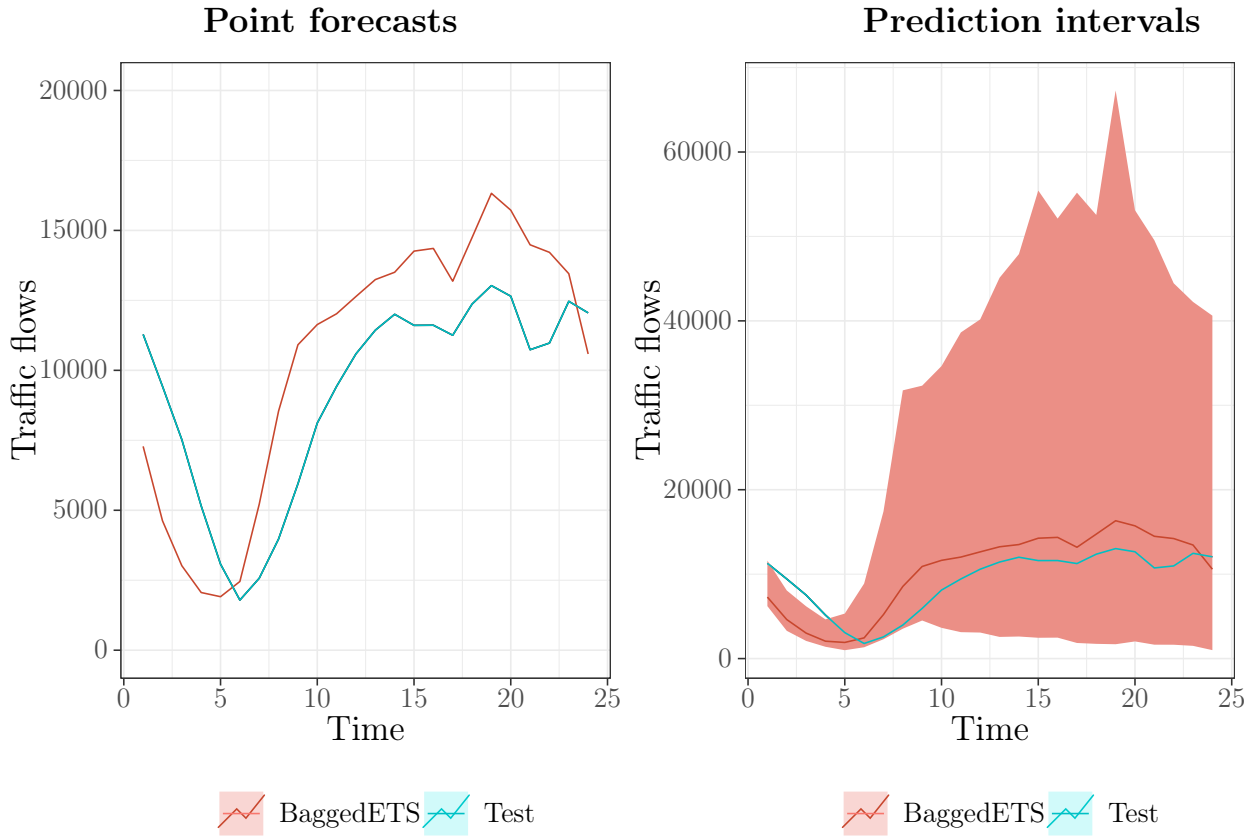

Figure 5.6: Point forecasts and simulated prediction intervals for the BaggedETS model

Notes: This figure shows the point forecasts from the BaggedETS ensemble model which are obtained by taking the median of 100 base learners. The simulated prediction intervals are shown separately in order to avoid scale distortions arising from disproportionately wide prediction intervals. The base learners are ETS models fitted to the bootstrapped training data. 95% prediction intervals are also shown which are obtained by generating 100 simulations from ETS models of the bootstrapped series, which have been refitted to the training data.

Lastly, we use a LGBM boosting ensemble model to boost regression trees. For the boosting ensemble model we employ a grid search cross validation to find the optimal combination of hyperparameters and the hyperparameter space considered is shown in Table 5.5. The final LGBM model setup is reported in Table 5.6.

Table 5.5: Hyperparameter space for the grid search CV - LGBM model (traffic)

Hyperparameter	**Values or options included in the gridsearch CV**
Colsample by tree	{ 0.8, 1.0 }
Min child weight	{ 3, 5 }
Learning rate	{ 0.1, 0.01, 0.001 }
Max depth	{ 5, 8 }
Reg_ lambda	{ 0.0, 0.7 }
Reg_ alpha	{ 0.0, 0.7 }
Grid search CV details	
Number of folds	3
Number of model candidates	96
Total number of model fits	288

Notes: This table contains the hyperparameter values included in the gridsearch CV for the LGBM model. *colsample_ by_ tree* sets the subsample ratio of input data columns used when constructing a tree. *min_ child_ weight* is the minimum sum of the instance weight needed in a child. The *learning_ rate* describes the boosting learning rate. *max_ depth* describes the maximum tree depth for base learners, which is the number of vertical levels with at least one split. *reg_ lambda* stands for the ℓ_2 regularization term on the weights and *reg_ alpha* represents a ℓ_1 regularization term on the weights. We use a "Tesla P100" graphical processing unit via Google Colab Pro to carry out this computationally intensive cross validation. Moreover, setting *n_ jobs* to -1 ensures that the maximum number of resources is used for parallel computational threads and parallel processing.

To conclude our analyses using traditional and ensemble models, we plot all 24 step ahead point forecasts obtained and the computed prediction intervals for the two traditional statistical models as well as those those obtained for the ensemble models against the test set. The plot grid is shown in Figure 5.7.

Table 5.6: Model setup for the LGBM ensemble (traffic)

Model Setup component	**Component details**
Prediction target	24 hours ahead traffic flows : $y_{T+1}, \ldots, y_{T+24}$
Input variables	History of 60 lags : $\{y_T, y_{T-1}, \ldots, y_{T-59}\}$
Output variables	Predictions of 24 steps ahead: $\{\hat{y}_{T+1}, \hat{y}_{t+2}, \ldots, \hat{y}_{T+24}\}$
Optimal training hyperparameters	
LGBM regressor ensemble model with 500 boosted trees	
Objective function	"Root mean squared error"
Subsample	1.0
N_ estimators	500
Num_ leaves	30
Min_ child_ samples	20
Colsample_ by_ tree	0.8
Min_ child_ weight	3
Learning_ rate	0.1
Max_ depth	8
Reg_ lambda	0.7
Reg_ alpha	0.0

Notes: This table contains all information about the model setup used for the LGBM ensemble model. The training hyperparameters are the ones obtained from the gridsearch cross validation plus the addtional hyperparameters that were fixed in advance. The fixed hyperparameters were set as follows: We choose the RMSE as an objective function and use a *LGBMRegressor* to boost regression trees. *Subsample* is the subsample ratio of the training example. *N_ estimators* stands for the number of boosted trees to fit. *Num_ leaves* is the maximum amount of tree leaves for each base model regression tree. *Min_ child_ samples* is the minimum amount of data needed in a child node. The other gridsearched hyperparameters are explained in **??**. The explanations of the hyperparameters are taken from the Python API of the lightgbm library (see https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMRegressor.html for further details).

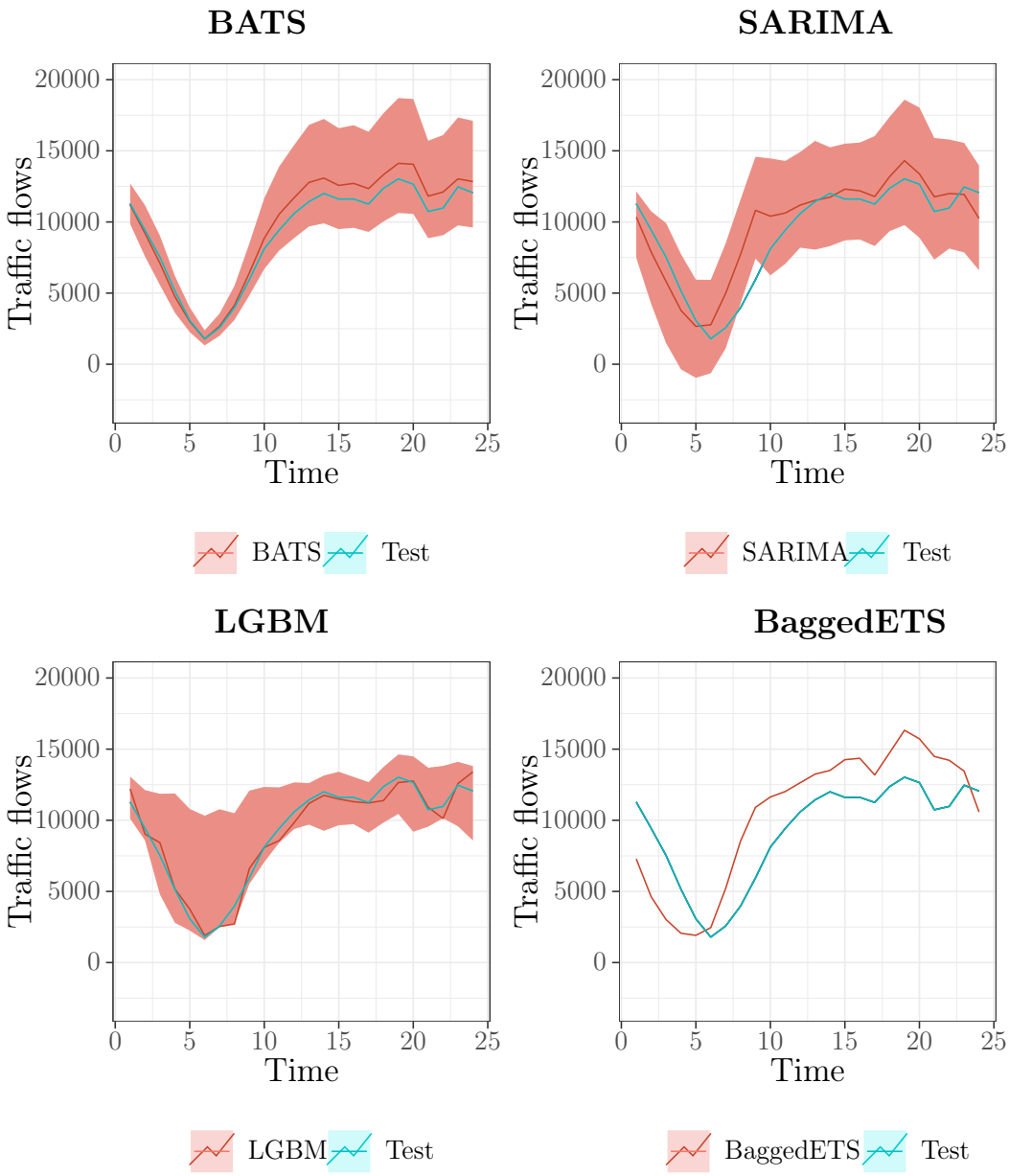

Figure 5.7: Comparison of point forecasts - Statistical and ensemble models (traffic)

Notes: This figure shows a grid plot of the 24 step ahead forecasts obtained from the traditional and the ensemble models. The top row shows the forecasts of the $BATS(0.035, \{4, 1\}, 0.934, 24, 168)$ model in the left panel as well as those of the SARIMA $(5, 0, 5) \times (4, 1, 1)_{24}$ model in the right panel. The bottom left panel depicts the forecasts from the LGBM boosting ensemble model and the bottom right panel shows the forecasts from the BaggedETS model. All point forecasts are plotted against the test set and with their corresponding prediction intervals except for the BaggedETS model Prediction intervals for the baggedETS model are left out in this gridplot in order to make the point forecasts more comparable but they are shown in Figure 5.6. LGBM prediction intervals are obtained through quantile regression techniques and the prediction intervals for the BATS and the SARIMA model are obtained through bootstrapping.

5.2 Traffic Forecasting Results from Neural Network Models

In this section, we describe the process of setting up and training our extended recurrent neural network models, the GRU and the LSTM. First, the grid search and the motivation behind the choices of certain values for the hyperparameters are explained. Moreover, we also consider a deep learning approach by stacking two hidden recurrent LSTM layers or GRU layers, respectively. Lastly, the best model for each architecture is chosen and compared against the traditional models.

Hyperparameter tuning for both the LSTM and the GRU neural networks is carried out using a grid search cross validation. Goodfellow et al. (2016) states that the common practice of a grid search CV consists of the following steps: First, a small set of values for each hyperparameter in question is selected. Second, a neural network model is trained for every possible combination of hyperparameters in the Cartesian product of these hyperparameter value sets in order to find the model with the lowest error measure on the validation set. In this way, the optimal combination of hyperparameter values can be found. However, Bengio (2012) states that there is no hard guarantee that minimizing the training error and evaluating the trained model on a validation set, which is used as a proxy for out of sample generalization, will lead to a low test error when the trained algorithm is fitted to new previously unseen examples from the data set. The main goal with respect to finding a good Machine Learning model is rather to find an optimal combination of hyperparameters such that the tendency of underfitting and overfitting is balanced and a good trade-off between bias and variance of the point estimates is achieved. The bias refers to the difference between the expectation of the point estimate over the data and the true value and the variance is a measure for the variation in the point estimate with alternative samplings from the training data. Goodfellow et al. (2016) state that **underfitting** refers to a situation in which both the training error and the test error are high and they characterize **overfitting** as a situation in which the training error is low but the test error is high as the algorithm fits the training data too well in order to generalize well when fit to out of sample data. The values of the hyperparameters used in the gridsearch CV are shown in Table 5.7.

Table 5.7: Hyperparameter search space - neural networks (traffic)

Hyperparameter	**Values or options included in the gridsearch CV**
Batch size	{ 100, 250, 500 }
Number of epochs	{ 250, 500, 750 }
Number of neurons	{ 500, 1000, 1500 }
Dropout	{ 0.0, 0.2, 0.4 }
Learning rate	{ 0.001, 0.01}
Optimizer	{'Adagrad', 'Adam', 'Adadelta' }
Grid search CV details	
Number of folds	3
Number of model candidates	486
Total number of model fits	1,458

Notes: This table contains the hyperparameter values included in the gridsearch CV for the LSTM and the GRU model. The *batch size* controls how often the weights of the neural network are updated. The *number of epochs* indicates how many times the entire training set of values is passed through all layers of the neural network. The *activation function* specifies the activation function used for the hidden layers (LSTM or GRU layer). The *number of neurons* sets the number of neurons used per hidden layer. *dropout* is the fraction of the neurons to drop for the linear transformation of the inputs, which can be used for regularization. The *learning rate* denotes the step size with which a step towards a local minimum in the opposite direction of the steepest gradient is undertaken. *Optimizer* refers to the implemented learning algorithm which is used to find the optimal network weights that minimize the chosen loss function. The descriptions of the hyperparameters were taken from the Tensorflow Python API (see https://www.tensorflow.org/api_docs/python/tf/keras/Model). The number of model candidates is given by the number of possible combinations among the hyperparameters, which is $(3C1)^5 * (2C1) = 3^5 * 2 = 486$.

The model setups used for both neural network architectures, which incorporate the results from the gridsearch CV are shown in Table 5.8. Following the heuristic approach stated by Hewamalage et al. (2019), who suggest to set the size of the input window to a value slightly bigger than the seasonal period, we opt for setting the size of the input window to 60, which is 2.5 times the seasonal period of 24.

Table 5.8: Model setup for the GRU and the LSTM neural networks (traffic)

Model Setup component	**Component details**	
Prediction target	24 hours ahead traffic flows : $y_{T+1}, \ldots, y_{T+24}$	
Input variables	History of 60 lags : $\{y_T, y_{T-1}, \ldots, y_{T-59}\}$	
Output variables	Predictions of 24 steps ahead: $\{\hat{y}_{T+1}, \hat{y}_{t+2}, \ldots, \hat{y}_{T+24}\}$	
Optimal training hyperparameters		
	GRU	**LSTM**
Activation function	'Leaky Relu'	'Leaky Relu'
Learning rate	0.001	0.01
Number of neurons	1500	1500
Batch size	100	250
Number of epochs	500	750
Optimizer	'Adam'	'Adagrad'
Dropout	0.0	0.0

Notes: This table contains all information about the model setup used for both the GRU and the LSTM neural networks. The training hyperparameters are the ones obtained from the respective grid search cross validation procedures which are carried out for each RNN architecture type separately.

Moreover, we also consider deep neural network architectures by adding a second recurrent hidden layer for each of the two network models. We stack two identical hidden layers using the hyperparameters shown in **??**.

Figure 5.8 shows the regular LSTM and the regular GRU neural network 24-step ahead forecasts plotted against the test set as well as those of the chosen Deep Long Short Term Memory neural network (DLSTM) and Deep Gated Recurrent Unit neural network (DGRU) models, both of which have two stacked hidden layers. We can see that the point forecasts produced by the LSTM and the GRU neural networks are fairly close in terms of their accuracy, while the regular GRU is lightly more accurate than the LSTM and the DLSTM is slightly more accurate than the DGRU model. Adding a second hidden layer leads to visibly more accurate predictions between 5 and 15 steps ahead.

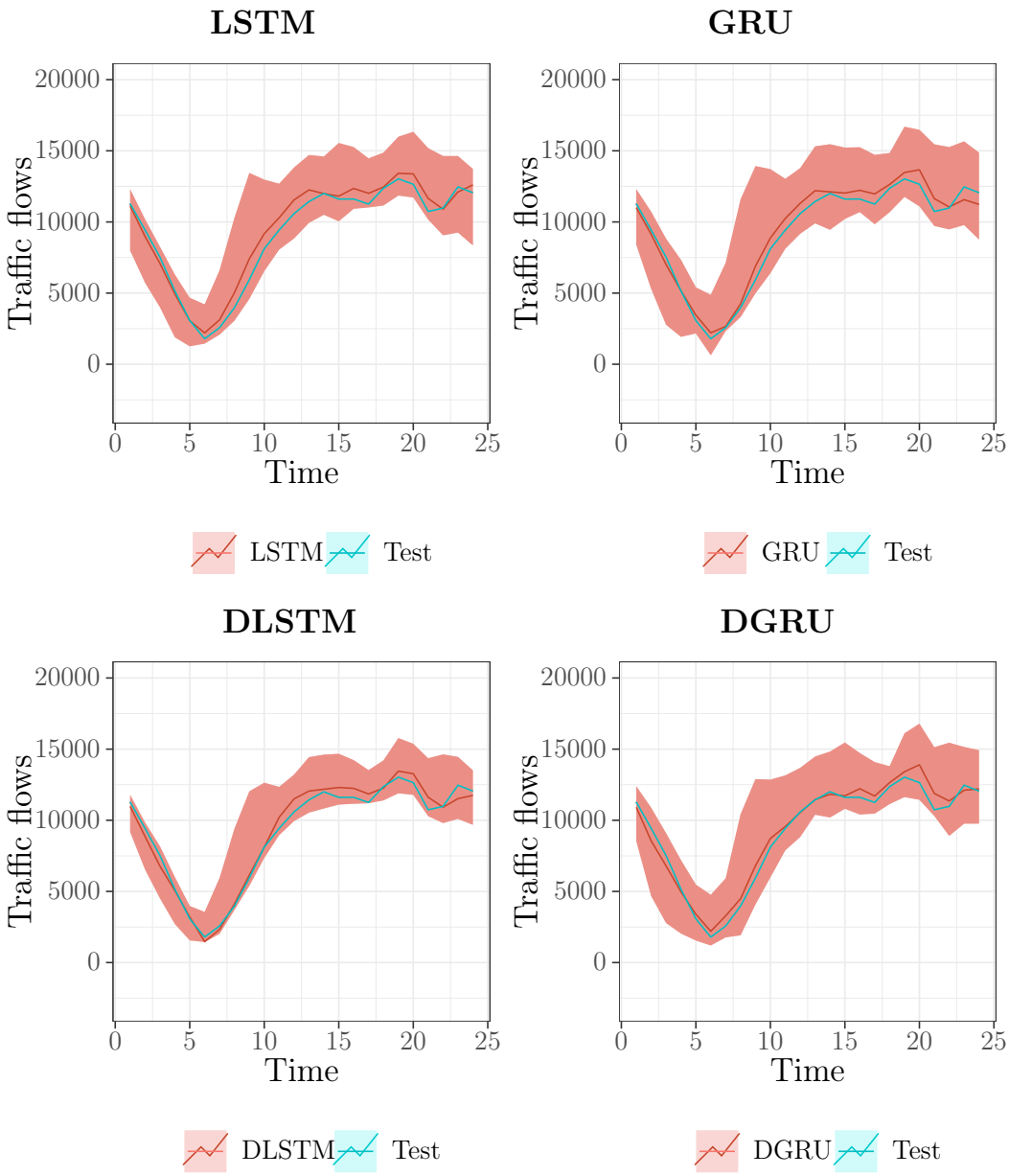

Figure 5.8: Comparison of point forecasts - Neural network models (traffic)

Notes: In the top row, this figure shows the 24 step ahead forecasts obtained from the regular GRU neural network and the regular LSTM neural network, each of which use only one hidden layer. In the bottom row, the forecasts of a deep LSTM neural network with 2 stacked hidden layers and those of a deep GRU neural network with 2 stacked hidden layers are shown. All forecasts shown in this figure are obtained using the MIMO approach and are plotted against the test set. Prediction intervals obtained through minimizing a quantile regression loss are also shown for all the network models.

Table 5.9 shows the test errors of the point forecasts obtained for all the forecasting models considered, i.e. the traditional models, the ensemble models and the neural network models. The Bagged ETS model can outperform the regular automatically chosen ETS traditional

model but not the other three benchmark models. The LGBM boosting ensemble model performs considerably better than all the benchmark models, whereas the BATS and TBATS models surprisingly produce divergent point forecasts. However, both ETS extension variants outperform all the benchmarks. Finally, both deep recurrent neural network models outperform all traditional and ensemble forecasting models as well as all the benchmarks. Among the the two considered deep learning models, the DLSTM with 2 stacked hidden LSTM layers emerges as the best forecasting model for the hourly traffic flow data overall with the DGRU model performing only marginally worse.

Table 5.10 shows a comparison of the three common evaluation metrics for prediction intervals computed for all the forecasting models. For the Machine Learning models, quantile regression techniques are used and for the traditional statistical models prediction intervals are computed using bootstrapping techniques to resample the model residuals. No distributional assumptions are required for the latter simulation technique but the model residuals are assumed to be uncorrelated. Due to computational constraints we set the number of simulated sample paths to 100 per model for which bootstrapped prediction intervals are required.
Taking into account the trade-off between coverage probability and mean width as measured by the CWC, the DGRU, LGBM and the BATS models produce similarly good prediction intervals. The PI measures of the BaggedETS model are considerably worse than those of both benchmarks and other candidate models. The DLSTM produces the best CWC score, which means that its PIs have the best trade-off between PICP and MPIW.

Table 5.9: Comparison of point forecast test error evaluation metrics (traffic)

Candidate Models							
Metric	**DLSTM**	**DGRU**	**LGBM**	**BaggedETS**	**SARIMA**	**BATS**	**TBATS**
RMSE	**528.05**	565.67	808.05	3056.02	1702.52	851.98	2125.12
MAE	**439.65**	461.54	611.96	2808.84	1310.50	729.58	1954.20
MAPE	**5.3336**	7.0332	9.3939	38.2627	22.4235	7.2395	27.2012
Benchmark Models							
Metric	**SNaive**	**ETS**	**SARIMA**	**FFNN**			
RMSE	3072.02	4425.75	2807.86	2492.64			
MAE	2348.29	3970.27	2127.95	1913.31			
MAPE	38.8332	56.7718	35.5932	34.5256			

Notes: This table shows a comparison of forecast error evaluation metrics for the point forecasts obtained for all the candidate and the benchmark models. The Machine Learning candidate models considered are the DLSTM and the DGRU neural networks as well as an LGBM ensemble model. The traditional statistical candidate models presented in this table are the manually refined SARIMA model, the BATS model and the TBATS model. Furthermore, as a hybrid model we consider the BaggedETS model.
As a first benchmark model we consider a seasonal naive forecasting model, which is obtained by setting each forecast equal to the last observed value from the same season, i.e. the value of the same hour of the day 24 hours before. We also consider an automated ETS(A, Ad, A) benchmark model obtained through the application of the *ets()* R function to the training set as well as an automated benchmark SARIMA$(5, 0, 5)(2, 1, 0)_{24}$ model obtained through the application of the *auto.arima()* R function to the training set. As a further benchmark, we also consider a non-recurrent FFNN model with 3 Dense layers containing 128, 64 and 24 neurons, respectively. The other fixed hyperparameters are as follows: *learning rate* = 0.01, *epochs* = 100, *batch size*=100, *loss*= MSE. The common forecast horizon is $h = 24$ time steps ahead for all models compared. The forecast error metrics are calculated on the test set.

Table 5.10: Comparison of prediction interval evaluation metrics (traffic)

Candidate Models							
Metric	**DLSTM**	**DGRU**	**LGBM**	**BaggedETS**	**SARIMA**	**BATS**	**TBATS**
PICP	**1.0**	**1.0**	**1.0**	0.8750	0.9167	**1.0**	**1.0**
MPIW	**3733.15**	5133.53	5128.74	32769.51	7432.07	5370.03	7981.65
CWC	**0.3322**	0.4568	0.4564	126.9055	4.1569	0.4778	0.7102

Benchmark Models				
Metric	**SNaive**	**ETS**	**SARIMA**	**FFNN**
PICP	0.9167	0.9167	0.9167	0.9583
MPIW	10345.00	15457.04	9717.26	7449.29
CWC	5.7862	8.6455	5.4351	0.6629

Notes: This table shows a comparison of prediction interval evaluation metrics for the point forecasts obtained for all the benchmark and the candidate models. The 95% prediction intervals for the DLSTM and the DGRU neural networks are obtained by fitting two respective neural networks with a quantile regression loss function to get the lower and upper bounds for the respective PI. Prediction intervals for the LGBM boosting model are also obtained by fitting two LGBM models with a quantile regression loss function. The PIs for all other candidate and benchmark models are computed by resampling the model residuals through bootstrapping. From 100 simulated sample paths we take the 0.025 and the 0.975 quantiles to obtain the lower and upper bounds of a 95% prediction interval. The *PICP* measure provides the coverage probability that indicates how many target values of the test set are contained in the interval spanned by the two PI bounds. The *MPIW* measure contains the average width of the PIs and the *CWC* quantifies the tradeoff between narrow PIs and a high coverage probability. For the CWC calculations, we use hyperparameter values of $\alpha = 0.95$ and $\xi = 50$. The NMPIW is calculated by dividing the MPIW by the range of the target variable, i.e. the range of the test set.

5.3 Robustness Checks For The Traffic Forecasting Results

This section contains our robustness checks with respect to variations in the forecast horizon (h), the weekday used as a test set as well as the amount of training data used to train the models. We only consider the best models obtained from every field, which are the BATS model, the LGBM ensemble model and the DLSTM neural network model. We also consider a Snaive benchmark model for comparison. The line plots showing the variation of the RMSE test error for every robustness check are shown in Figure 5.9.

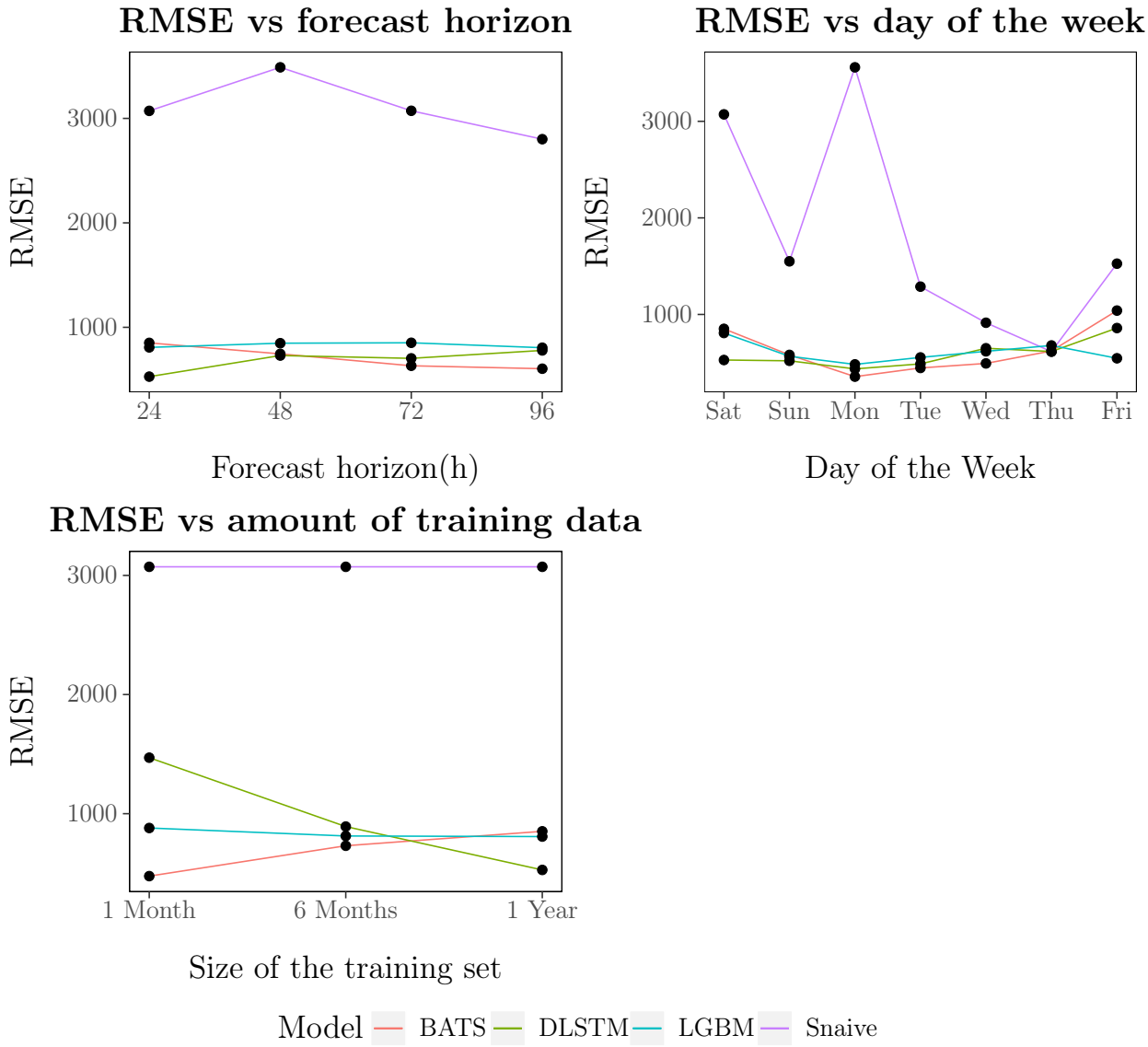

Figure 5.9: Robustness checks w.r.t. forecast horizon, weekday and train size

Notes: This grid plot shows the robustness checks for the traffic data. The top left panel shows the RMSE values as the forecast horizon is varied from h=24 hours ahead to h=96 hours ahead. The top right panel shows the RMSE values as the weekday used as a 24 hour test set is varied. The different weekdays are taken from the held out test data. The bottom panel shows the RMSE values as the training data used to train the models is varied from 1 month to 1 year.

The top left panel of Figure 5.9 shows a robustness check with respect to the length of the forecast horizon. According to the RMSE test errors, all candidate models outperform the seasonal naive benchmark model across all forecast horizons. The test error of the LGBM model stays roughly constant and the test errors for the BATS and the DLSTM model fluctuate within a range of 250. Interestingly, the BATS model's RMSE decreases as the forecast horizon

increases, whereas the RMSE of the DLSTM increases. Both Machine Learning models require different preprocessing and setting up different models, which have to be retrained per forecast horizon. The number of input lags stays the same for all the models but changes in the desired output dimensions require retraining the Machine Learning models. The DLSTM creates forecasts for a target window equal to the number of steps ahead simultaneously and the boosting ensemble forecasts consist of multi-target regression, i.e. fitting one LGBM regression model for every step ahead. The BATS model forecasts are obtained recursively from the same initial model and do not require retraining. Note that further fine tuning of the hyperparameters of the Machine Learning models for the additional three forecast horizons has not been carried out for this robustness analysis.

The top right panel of Figure 5.9 shows a robustness check with respect to the weekday represented by the 24 hours used as a test set to explore sensitivity of the models with respect to specific weekdays. The benchmark SNAIVE model only provides accurate point forecasts which are on the same level as those of the candidate models on Thursday. On all other weekdays, all candidate models perform considerably better. A key insight from this plot is that no one forecasting model is able to outperform all other models on every given weekday. However, the DLSTM model has the lowest average RMSE value across the entire week of 585, closely followed by the average RMSE values of the LGBM and the BATS, which are 608 and 627, respectively.

Lastly, we conduct a robustness check on the amount of training data used to train the models, the results of which are shown in the bottom panel of Figure 5.9. By design, the SNAIVE benchmark model produces indentical RMSE values due to the fact that only the last seasonal period is used to compute the predictions. More interestingly, The DLSTM model gets more accurate as the amount of training data is increased, whereas the BATS model's accuracy decreases when using more training data. The LGBM model's accuracy remains roughly constant. All the models require retraining for the different training sizes considered.

5.4 EMS Forecasting Results from Traditional and Ensemble Models

We apply the same procedure to obtain a manual SARIMA model as in Section 5.1. The ACF and PACF plots from the automatically chosen and the manual model are shown in Figure 5.10.

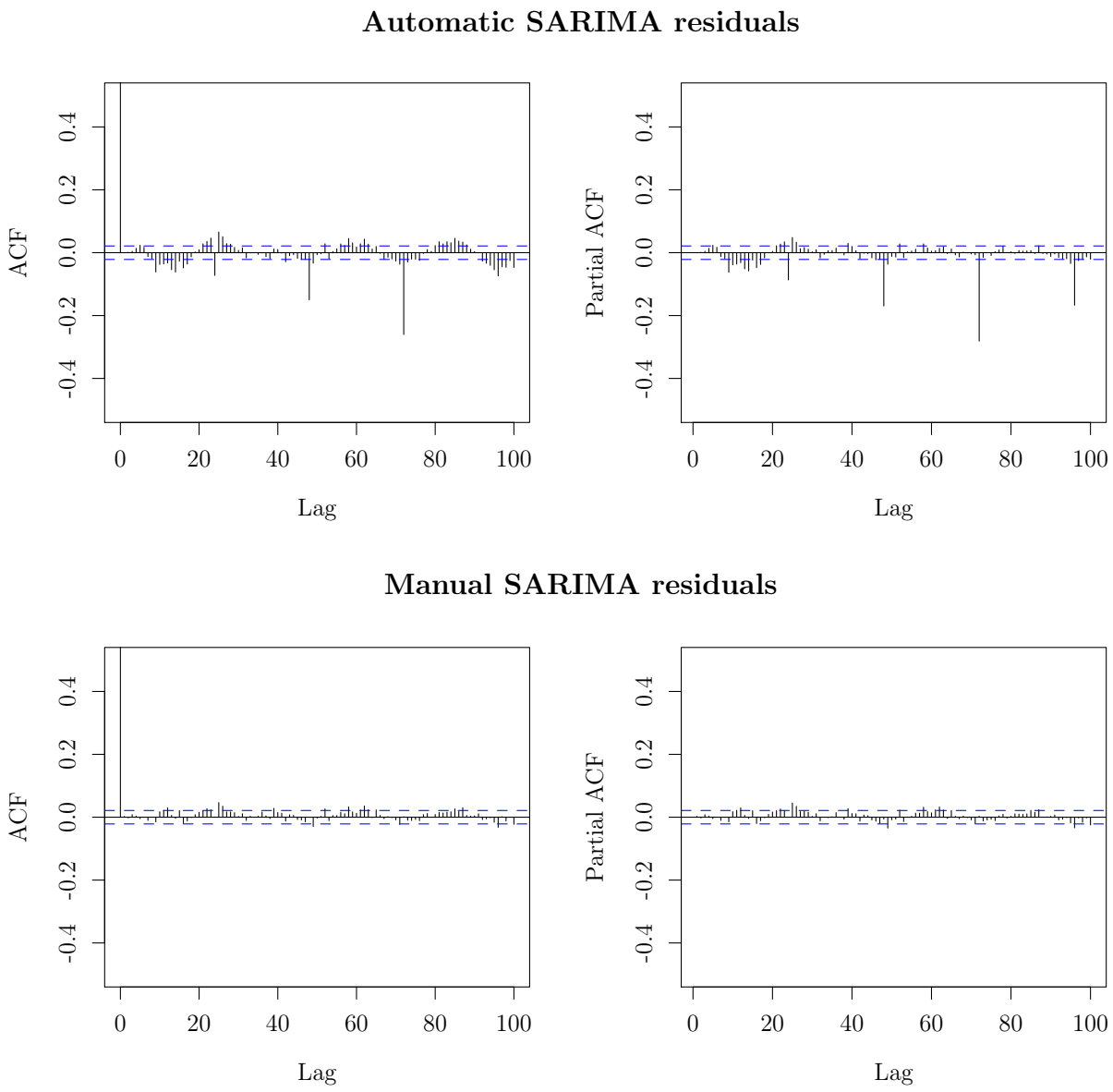

Figure 5.10: ACF/PACF plot comparison of SARIMA model residuals (EMS)

Notes: This figure shows the autocorrelation and the partial autocorrelation for up the 100 lags of the automatically chosen SARIMA $(4,0,0)(2,1,0)_{24}$ model's residuals, the manually refined SARIMA $(4,0,0)(3,1,1)_{24}$ model's residuals and the corresponding confidence bounds at the 95 % level.

It is clearly visible how the autocorrelation at the seasonal lags is drastically reduced through the manual modeling procedure. The chosen manual SARIMA model contains one additional seasonal AR lag and one extra seasonal MA lag. Point forecasts and bootstrapped prediction intervals based on 100 simulated sample paths are shown in Figure 5.11.

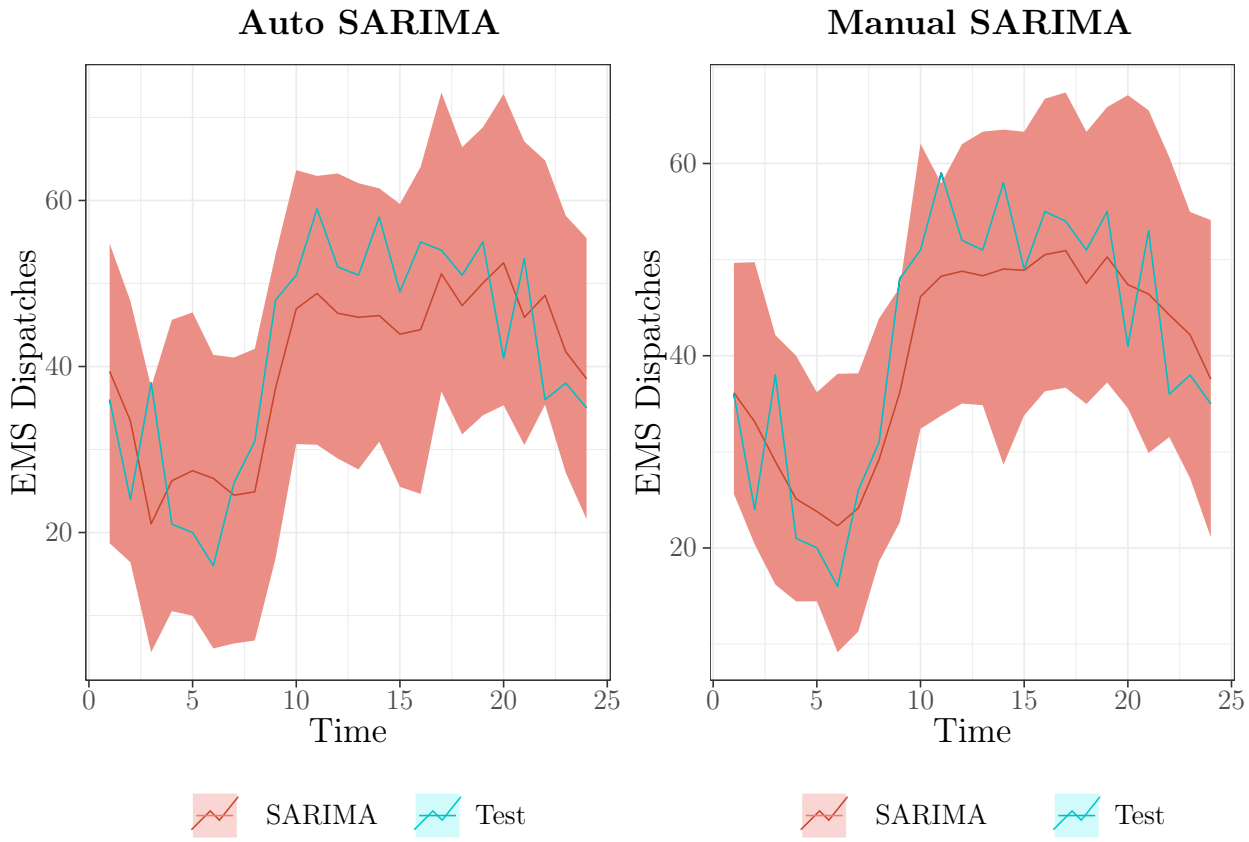

Figure 5.11: Comparison manual vs. automatic SARIMA model (EMS)

Notes: This figure shows the 24 step ahead predictions as well as the 95% prediction intervals of the automatically chosen SARIMA $(4,0,0) \times (2,1,0)_{24}$ model plotted against those from the manually refined SARIMA $(4,0,0) \times (3,1,1)_{24}$ model, both of which are compared against the test set.

Table 5.11: SARIMA model residual diagnostic tests (EMS)

Autocorrelation				
	$\text{SARIMA}(4,0,0) \times (2,1,0)_{24}$		$\text{SARIMA}(4,0,0) \times (3,1,1)_{24}$	
Max lag	**LB test stat(Q^*)**	**P-value**	**LB test stat(Q^*)**	**P-value**
1	0.0018	0.9657	0.0632	0.8015
2	0.0194	0.9657	0.0999	0.9513
3	0.1962	0.9782	0.6482	0.8853
24	251.19	$< 0.0000^{***}$	43.284	0.0092^{***}
48	544.14	$< 0.0000^{***}$	98.17	0.0000^{***}
72	1234.80	$< 0.0000^{***}$	159.00	0.0000^{***}
Normality				
	SARIMA4, 0, 0) $\times (2,1,0)_{24}$		$\text{SARIMA}(4,0,0) \times (3,1,1)_{24}$	
	JB test stat	**P-value**	**JB test stat**	**P-value**
	24.017	$< 0.0000^{***}$	27.034	$< 0.0000^{***}$
	Skewness	**Kurtosis** -3	**Skewness**	**Kurtosis** -3
	0.0875	0.1945	0.1011	0.1905
Conditional Heteroskedasticity				
	$\text{SARIMA}(4,0,0) \times (2,1,0)_{24}$		$\text{SARIMA}(4,0,0) \times (3,1,1)_{24}$	
Max lag	**LM test stat**	**P-value**	**LM test stat**	**P-value**
4	2,479	$< 0.0000^{***}$	2,583	$< 0.0000^{***}$
12	821	$< 0.0000^{***}$	853	$< 0.0000^{***}$
24	405	$< 0.0000^{***}$	420	$< 0.0000^{***}$

Notes: This table shows the test statistics and p-values for the Ljung-Box (LB) test for autocorrelation among the residuals, the Jarque-Bera (JB) test for normality and the Engle Lagrange Multiplier test for conditional heteroskedasticity of the squared model residuals for both the automatically chosen and the manually chosen SARIMA model. The automatically chosen model is an SARIMA $(4,0,0)(2,1,0)_{24}$ model and the manually refined model is an SARIMA $(4,0,0)(3,1,1)_{24}$ model. The level of significance is given by *: p-value < 0.10; **: p-value < 0.05; ***: p-value < 0.01.

Next, we fit both a BATS and a TBATS model to the EMS training data modeling daily and weekly seasonal periods explicitly. Neither model selects a Box-Cox transformation but the BATS model involves a damped trend while the TBATS model includes ARMA(4,4) errors. Point forecasts and bootstrapped prediction intervals are shown in Figure 5.12. The two models exhibit point forecast accuracy and PI quality which are closely matched in contrast to forecasting performance on the traffic data set.

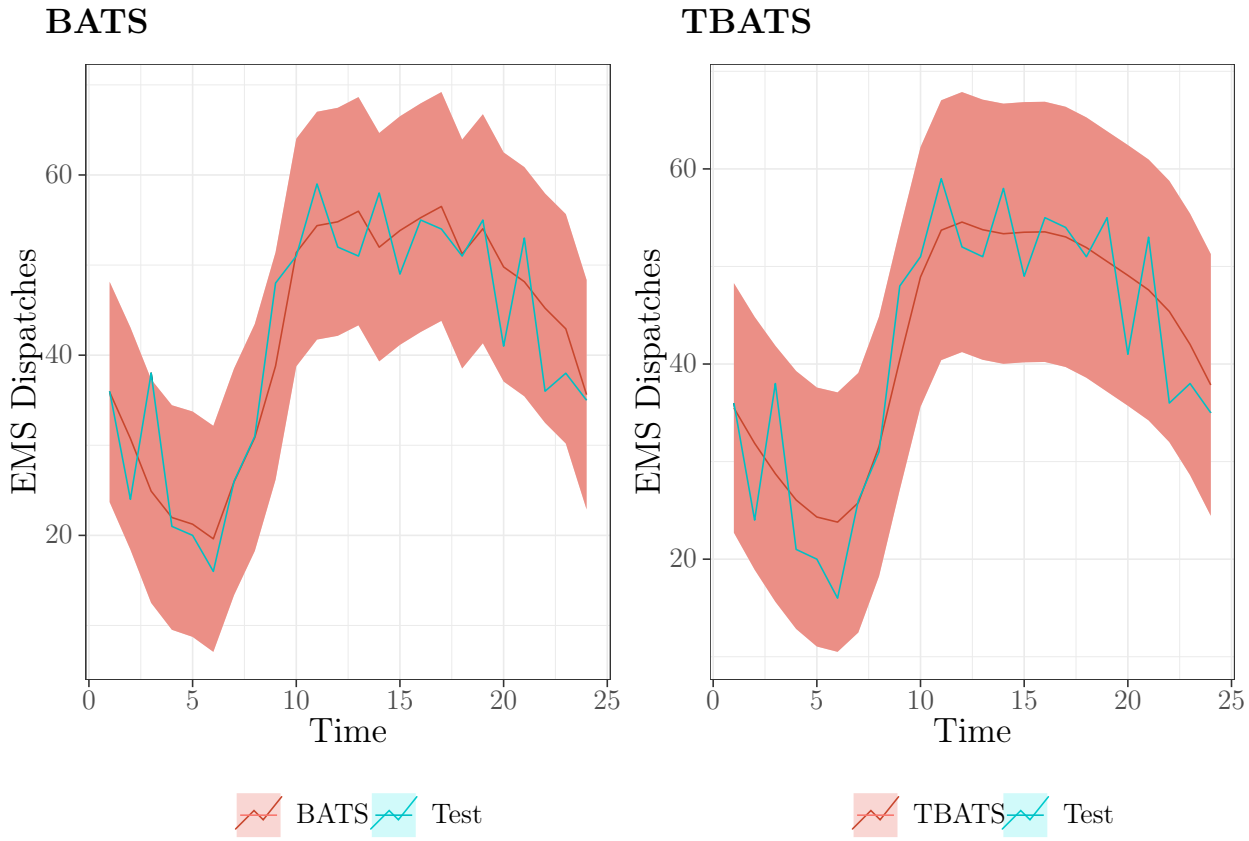

Figure 5.12: Comparison BATS vs TBATS model (EMS)

Notes: This figure shows the 24 step ahead predictions as well as the 95% prediction intervals of the $BATS(1, \{0,0\}, 0.833, 24, 168)$ model plotted against those from the $TBATS(1, \{4,4\}, -, <24,6>, <168,6>)$ model, both of which are compared against the test set.

Finally, we provide a visual comparison of all the point forecasts of all the chosen candidate models from the traditional statistical and the ensemble methods field, which is shown in **??**. We can see that the ETS extensions generalize better when fitted to the out of sample test data than the ensemble models. The simulated prediction intervals for the BaggedETS model are too narrow to cover the entire test data.

?? shows the hyperparameter space used for the gridsearch cross validation employed to develop

the LGBM boosting ensemble model for the EMS data. **??** shows the chosen combination of hyperparameters used for the EMS data to build an LGBM ensemble model.

Table 5.12: Hyperparameter space for the grid search CV - LGBM model (EMS)

Hyperparameter	**Values or options included in the gridsearch CV**
Colsample by tree	{ 0.8, 1.0 }
Learning rate	{ 0.1, 0.01 }
Max depth	{ 8, 15 }
N_ estimators	{ 100, 250 }
Num_ leaves	{ 100, 200 }
Reg_ lambda	{ 0.0, 0.7 }
Reg_ alpha	{ 0.0, 0.7 }
Grid search CV details	
Number of folds	3
Number of model candidates	128
Total number of model fits	384

Notes: This table contains the hyperparameter values included in the gridsearch CV for the LGBM model. *colsample_ by_ tree* sets the subsample ratio of input data columns used when constructing a tree. The *learning_ rate* describes the boosting learning rate. *max_ depth* describes the maximum tree depth for base learners, which is the number of vertical levels with at least one split. *reg_ lambda* stands for the ℓ_2 regularization term on the weights and *reg_ alpha* represents a ℓ_1 regularization term on the weights. We use a "Tesla P100" graphical processing unit via Google Colab Pro to carry out this computationally intensive cross validation. Moreover, setting *n_ jobs* to -1 ensures that the maximum number of resources is used for parallel computational threads and parallel processing.

Table 5.13: Model setup for the LGBM ensemble (EMS)

Model Setup component	**Component details**
Prediction target	24 hours ahead dispatches : $y_{T+1}, \ldots, y_{T+24}$
Input variables	History of 60 lags : $\{y_T, y_{T-1}, \ldots, y_{T-59}\}$
Output variables	Predictions of 24 steps ahead: $\{\hat{y}_{T+1}, \hat{y}_{t+2}, \ldots, \hat{y}_{T+24}\}$
Optimal training hyperparameters	
LGBM regressor ensemble model with 250 boosted trees	
Objective function	"Root mean squared error"
Subsample	1.0
N_ estimators	250
Num_ leaves	100
Min_ child_ samples	20
Colsample_ by_ tree	0.8
Learning_ rate	0.01
Max_ depth	15
Reg_ lambda	0.0
Reg_ alpha	0.0

Notes: This table contains all information about the model setup used for the LGBM ensemble model. The training hyperparameters are the ones obtained from the gridsearch cross validation plus the addtional hyperparameters that were fixed in advance. The fixed hyperparameters were set as follows: We choose the RMSE as an objective function and use a *LGBMRegressor* to boost regression trees. *Subsample* is the subsample ratio of the training example. *N_ estimators* stands for the number of boosted trees to fit. *Num_ leaves* is the maximum amount of tree leaves for each base model regression tree. *Min_ child_ samples* is the minimum amount of data needed in a child node. The other gridsearched hyperparameters are explained in **??**. The explanations of the hyperparameters are taken from the Python API of the lightgbm library (see https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMRegressor.html for further details).

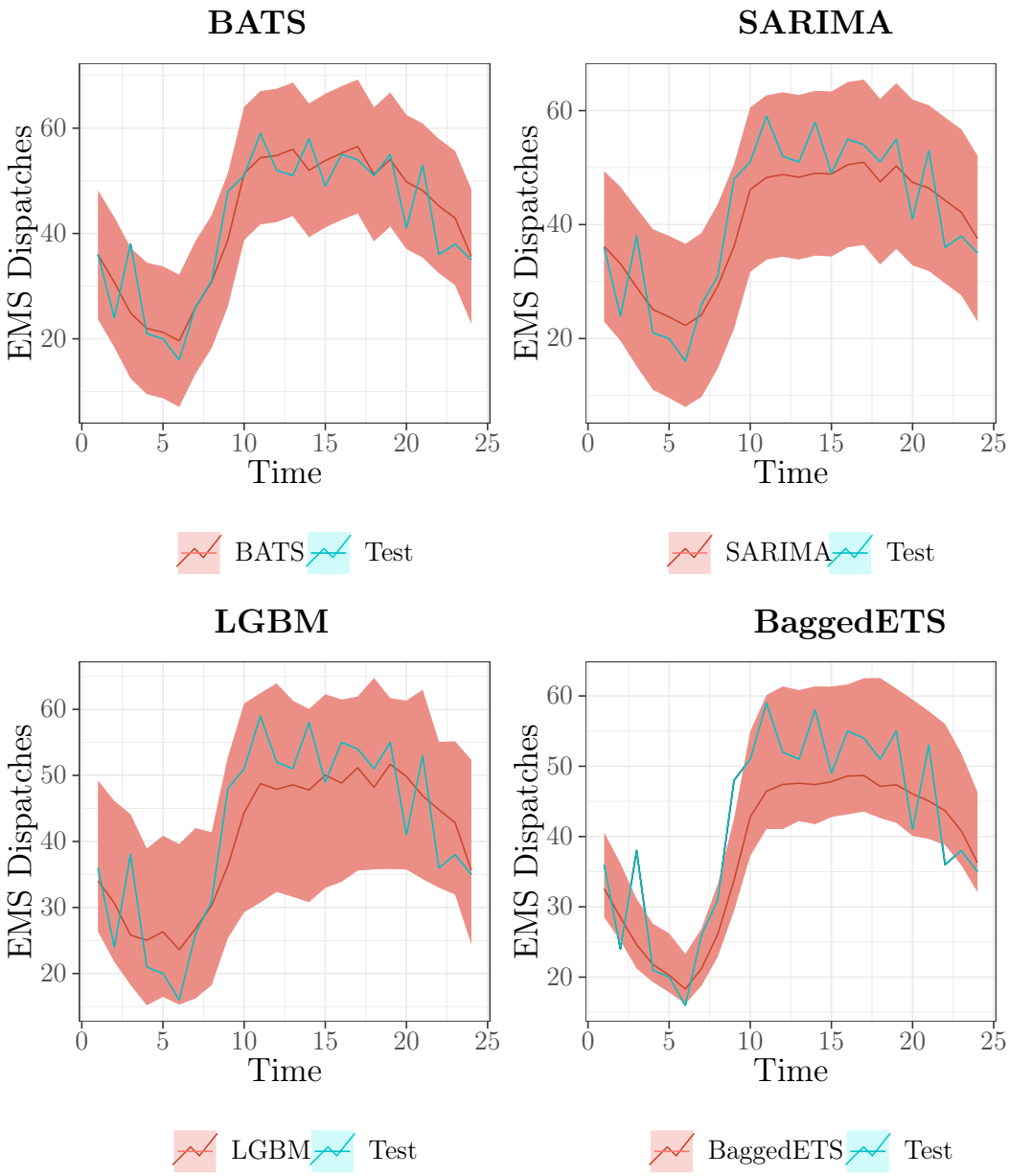

Figure 5.13: Comparison of point forecasts - Statistical and ensemble models (EMS)

Notes: This figure shows a grid plot of the recursive 24 step ahead forecasts obtained from the traditional and ensemble models. The top row shows the forecasts of the $BATS(1, 0, 0, 0.833, 24, 168)$ model in the left panel as well as those of the SARIMA $(4, 0, 0) \times (3, 1, 1)_{24}$ model in the right panel. The bottom left panel depicts the LGBM boosting ensemble model forecasts and the bottom right panel shows the forecasts from the BaggedETS model. All point forecasts are plotted against the first 24 observations of the test set and with their corresponding prediction intervals.

5.5 EMS Forecasting Results from Neural Network Models

Since no optimal set of hyperparameters can be reused to train neural networks that generalize well across two completely different data sets, we set up a new search space to find the optimal hyperparameter combination for each of the recurrent neural network architectures for the EMS demand data. Due to the observation that the neural network models initially seemed to be underfitting, we increase the width of the network by increasing the number of neurons in the hidden layer and the number of training instances, i.e. the number of epochs. We choose a smaller grid compared to that used for the traffic flow data due to computational considerations as wider network models require more computation time.

Table 5.14: Hyperparameter search space - neural networks (EMS)

Hyperparameter	**Values or options included in the gridsearch CV**
Batch size	{ 150, 250 }
Number of epochs	{ 750, 1000 }
Number of neurons	{ 1500, 2000 }
Learning rate	{ 0.01, 0.001, 0.0001}
Optimizer	{'Adam', 'Adadelta' }
Grid search CV details	
Number of folds	3
Number of model candidates	48
Total number of model fits	144

Notes: This table contains the hyperparameter values included in the gridsearch CV for the LSTM and the GRU model. The *batch size* controls how often the weights of the neural network are updated. The *number of epochs* indicates how many times the entire training set of values is passed through all layers of the neural network. The *activation function* specifies the activation function used for the hidden layers (LSTM or GRU layer). The *number of neurons* sets the number of neurons used per layer. *dropout* is the fraction of the neurons to drop for the linear transformation of the inputs, which can be used for regularization. The *learning rate* denotes the step size with which a step towards a local minimum in the opposite direction of the steepest gradient is undertaken. *Optimizer* refers to the implemented learning algorithm which is used to find the optimal network weights that minimize the chosen loss function. The descriptions of the hyperparameters were taken from the Tensorflow Python API (see https://www.tensorflow.org/api_docs/python/tf/keras/Model). The number of model candidates is given by the number of possible combinations among the hyperparameters, which is $(3C1) * (2C1)^4 = 3 * 16 = 48$.

?? shows the hyperparameter search space and **??** contains the all gridsearched and fixed hyperparameters chosen to fit the LSTM and GRU models.

Table 5.15: Model setup for the GRU and the LSTM neural networks (EMS)

Model Setup component	**Component details**	
Prediction target	24 hours ahead dispatches : $y_{T+1}, \ldots, y_{T+24}$	
Input variables	History of 60 lags : $\{y_T, y_{T-1}, \ldots, y_{T-59}\}$	
Output variables	Predictions of 24 steps ahead: $\{\hat{y}_{T+1}, \hat{y}_{t+2}, \ldots, \hat{y}_{T+24}\}$	
Optimal training hyperparameters		
	GRU	**LSTM**
Activation function	'Leaky Relu'	'Leaky Relu'
Learning rate	0.01	0.001
Number of neurons	2000	1500
Batch size	150	168
Number of epochs	750	750
Optimizer	'Adadelta'	'Adadelta'
Dropout	0.0	0.0

Notes: This table contains all information about the model setup used for both the GRU and the LSTM neural networks. The training hyperparameters are the ones obtained from the respective grid search cross validation procedures which are carried out for each RNN architecture type separately.

?? finally shows a visual comparison of the point forecasts and prediction intervals for the regular and deep neural network models. It can be seen that the GRU as well as the DGRU with two stacked hidden layers produce point forecasts that generalize slightly better to the test data than the LSTM models do.

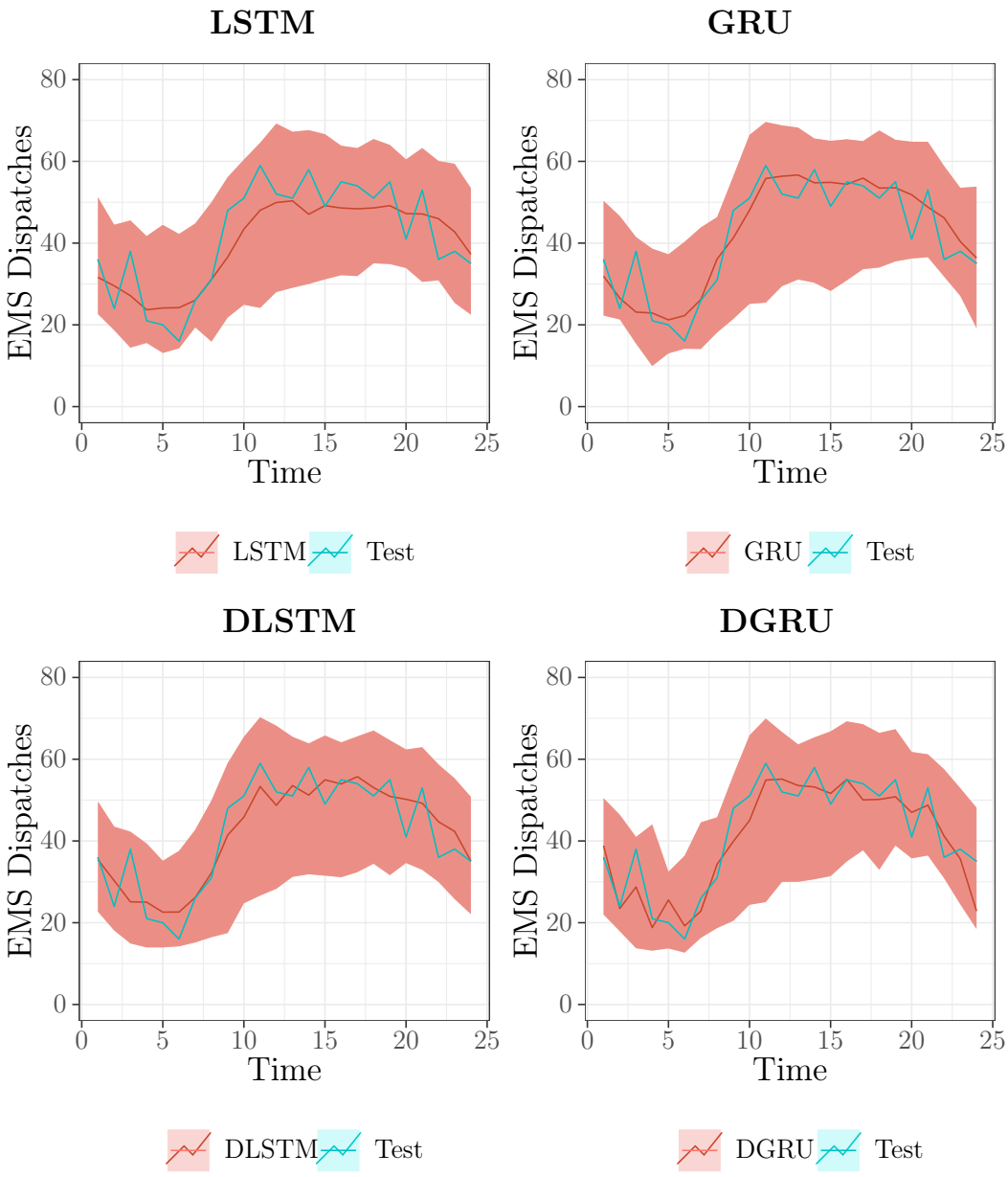

Figure 5.14: Comparison of point forecasts - neural networks (EMS)

Notes: In the top row, this figure shows the 24 step ahead forecasts obtained from the regular GRU neural network and the regular LSTM neural network, each of which use only one hidden layer. In the bottom row, the forecasts of a deep LSTM neural network with 2 stacked hidden layers and those of a deep GRU neural network with 2 stacked hidden layers are shown. All forecasts shown in this figure are obtained using the MIMO approach and are plotted against the test set. Prediction intervals obtained through minimizing a quantile regression loss are also shown for all the network models.

A numerical comparison of the test error metrics obtained for all candidate models and the benchmark models is given in **??**. The PIs of the BaggedETS model and the manual ARIMA model provide the lowest values by all three metrics for coverage, mean width and a combination

thereof. The best prediction intervals can be obtained for the ETS extension models.

Table 5.16: Comparison of point forecast test error evaluation metrics (EMS)

Candidate Models							
Metric	**DLSTM**	**DGRU**	**LGBM**	**BaggedETS**	**SARIMA**	**BATS**	**TBATS**
RMSE	5.3765	**4.9751**	6.4795	6.8838	5.9832	5.1989	5.1236
MAE	4.3909	4.1830	5.4597	5.7131	5.0943	**3.7984**	4.2745
MAPE	12.0963	11.2779	14.8491	13.4602	13.7923	**9.9314**	12.5473
Benchmark Models							
Metric	**SNaive**	**ETS**	**SARIMA**	**FFNN**			
RMSE	11.0189	6.1851	8.1699	7.5094			
MAE	9.4166	5.1224	7.2306	6.2397			
MAPE	24.1956	12.7362	20.1092	19.4061			

Notes: This table shows a comparison of forecast error evaluation metrics for the point forecasts obtained for all the candidate and the benchmark models. The Machine Learning candidate models considered are the DLSTM and the DGRU neural networks as well as an LGBM ensemble model. The traditional statistical candidate models presented in this table are the manually refined SARIMA model, the BATS model and the TBATS model. Furthermore, as a hybrid model we consider the BaggedETS model.
As a first benchmark model we consider a seasonal naive forecasting model, which is obtained by setting each forecast equal to the last observed value from the same season, i.e. the value of the same hour of the day 24 hours before. We also consider an automated ETS(A, Ad, A) benchmark model obtained through the application of the *ets()* R function to the training set as well as an automated benchmark SARIMA$(5, 0, 5)(2, 1, 0)_{24}$ model obtained through the application of the *auto.arima()* R function to the training set. As a further benchmark, we also consider a non-recurrent FFNN model with 3 Dense layers containing 128, 64 and 24 neurons, respectively. The other fixed hyperparameters are as follows: *learning rate* = 0.01, *epochs* = 100, *batch size*=100, *loss*= MSE. The common forecast horizon is $h = 24$ time steps ahead for all models compared. The forecast error metrics are calculated on the test set.

A Numerical comparison of the prediction interval metrics is presented in **??**.

Table 5.17: Comparison of prediction interval evaluation metrics (EMS)

Candidate Models							
Metric	**DLSTM**	**DGRU**	**LGBM**	**BaggedETS**	**SARIMA**	**BATS**	**TBATS**
PICP	**1.0**	**1.0**	**1.0**	0.9167	0.9167	0.9583	**1.0**
MPIW	31.3929	30.7371	26.6975	30.0278	28.4155	**25.2107**	26.5626
CWC	0.7301	0.7148	0.6209	4.3894	4.1537	**0.5863**	0.6177
Benchmark Models							
Metric	**SNaive**	**ETS**	**SARIMA**	**FFNN**			
PICP	0.9583	1.0	0.9583	1.0			
MPIW	38.8648	40.4169	34.3559	33.4885			
CWC	0.9038	0.9399	0.7990	0.7788			

Notes: This table shows a comparison of prediction interval evaluation metrics for the point forecasts obtained for all the benchmark and the candidate models. The 95% prediction intervals for the DLSTM and the DGRU neural networks were obtained by fitting two respective neural networks with a quantile regression loss function to get the lower and upper bounds for the respective PI. Prediction intervals for the LGBM boosting model were also obtained by fitting two LGBM models with a quantile regression loss function. The 95 % PIs for the ETS and the bagged ETS model were obtained through bootstrapping. The *PICP* measure provides the coverage probability that indicates how many target values of the test set are contained in the interval spanned by the two PI bounds. The *MPIW* measure contains the average width of the PIs and the *CWC* quantifies the tradeoff between narrow PIs and a high coverage probability. For the CWC calculations, we use hyperparameter values of $\alpha = 0.95$ and $\xi = 50$. The NMPIW is calculated by dividing the MPIW by the range of the target variable, i.e. the range of the test set.

Section 6

Discussion

Research space. In this paper, we limit our analyses to univariate time series forecasting incorporating only a time dimension. We also consider high-frequency data as both of our data sets are resampled to an hourly frequency which results in time series with 8,712 observations in case of the traffic flow data and 8,760 in case of the EMS demand data. It is also important to point out that we are focusing our analyses on two very specific individual time series (local modeling) from a smart city context within the traffic forecasting and EMS demand forecasting domains. Considering a set of multiple time series as the one used in the M forecasting competitions, on the other hand, would result in a different forecasting task (global modeling), which might require to come up with a forecasting model that can learn time series properties across multiple time series. According to Makridakis et al. (2020), the M4 forecasting competition includes 100,000 time series from a variety of disjoint domains such as Demographics or Finance for example and was revised in comparison to previous forecasting competitions in the sense that high-frequency data comprising hourly, daily and weekly frequencies were also included in addition to the common low-frequency data that include monthly, quarterly and yearly frequencies. A further novelty that was incorporated in the M4 competition for the first time is the requirement of providing prediction intervals to get an idea of the uncertainty associated with the obtained point forecasts. This paper addresses this development by providing prediction intervals for all forecasting models considered in this paper from both the statistical and the Machine Learning field.

Limitations. As this paper places a high emphasis on the theoretical underpinnings and the mathematical way of parameter estimation for the respective models, practical implications are only considered to a very limited extend and a comparison to industry forecasting methodology standards is not provided. This paper carries out a comparison of a pre-selected spectrum of forecasting models and several further models are not part of the scope of this paper. For instance, Makridakis, Spiliotis, and Assimakopoulos (2018) mention Bayesian Neural Networks

and K nearest neighbors regression, which are not part of the models considered in this paper. Furthermore, Hewamalage et al. (2019) investigate the modeling of seasonality especially in the context of neural networks and find mixed evidence as to whether deseasonalization leads to significantly different forecasting results. They also argue that neural networks are able to model seasonality if multiple series have similar seasonal patterns. Since we only consider an individual time series at a time, we do not investigate a deseasonalization of the data further for the purpose of this paper. Moreover, we do not investigate the computational efficiency of the training process of Machine Learning models and in particular neural networks. Livni, Shalev-Shwartz, and Shamir (2014) separate neural network architectures into hypothesis classes which are characterized by a directed acyclic graph which represents the network and an activation function. They define a hypothesis class as the set of all prediction rules which can be obtained by using the same neural network architecture while altering the weights and investigate the training time and the number of training samples required to learn a hypothesis class. Lastly, for the training of all Machine Learning models considered in this paper we use a NVIDIA Tesla P100 GPU[1] to which we obtain access via the cloud computing services offered via a Google Colab Pro subscription, which comes with the restriction of a maximum access limit of 24 hours per cloud based runtime session. While this GPU is powerful enough to run a sophisticated grid search to find the best hyperparameters for our boosting ensemble and neural network prediction models, the time limit per runtime session puts a limit on our possibilities to train the Machine Learning models.

Further research. Bergstra and Bengio (2012) show empirically that hyperparameter optimization for Machine Learning models and neural networks specifically can be carried out more efficiently than with a grid search through a random search of hyperparameters as dimensionality of the hyperparameter space increases. They argue that a random search can be more efficient due to the fact that not all hyperparameters are equally important to tune. It follows from that line of reasoning that the grid search procedure might allocate too many model fit trials to dimensions that are not important. Future work could therefore leverage these findings in order to improve the neural network forecasting results even further.
Another way of expanding the analyses found in this paper is to employ forecasting models that include a spatial dimension. Safikhani, Kamga, Mudigonda, Faghih, and Moghimi (2020) also predict demand for Yellow Cabs in New York City but they incorporate spatial variation by ZIP code as well as temporal variation measured in 15 minute intervals. They employ a spatio-temporal STARMA model as well as a double hierarchical group Least absolute shrinkage and selection operator (LASSO) model that allows for the penalization of parameters the

[1]See https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf for technical details about the Tesla P100 GPU used by Google.

further away they are temporally and spatially.
A further direction future research could take is the usage of the boosting procedure proposed by Taieb and Hyndman (2014), which addresses the forecast horizon related bias and estimation variance trade-off faced when choosing between recursive one-step ahead forecasts and direct multi-step ahead forecasts. The authors propose a boosting autoregression procedure for the residuals of the recursive linear forecasts from an AR model that is applied at every time horizon to carry out small nonlinear adjustments. Robustness checks on prediction intervals would be useful.

Main findings. When first examining the two chosen high-frequency data sets we detected multiple levels of seasonality in both cases. We expected the Machine Learning models to be able to cope with these data characteristics due their flexibility with respect to modeling nonlinear relationships (Hewamalage et al., 2019). However, the oldest traditional models, the ETS and the SARIMA model, are not able to model multiple seasonal patterns by design. Thus, we included the more recently proposed TBATS model in the comparison, which is the only model from the traditional field which can account for multiple levels of seasonality. The BaggedETS model which allows for the application of the bagging technique to a time series involves a STL decomposition as a part of the bootstrapping procedure but it does not allow for multiple levels of seasonality. When we compare point forecasts across all models from both fields we find that the models which can model data that exhibit multiple seasonal patterns perform better. The most accurate point forecasts for our traffic data are obtained by deep recurrent neural network models. We find that all prediction intervals obtained through quantile regression yield a higher coverage probability and narrower PI widths than those produced by bootstrapping techniques or analytical derivations based on distributional assumptions. The lower coverage probabilities obtained for the SARIMA and ETS models could arise because neither the no autocorrelation assumption nor the normality assumption is met. The bootstrapping technique produces comparable PI results to those of the ML models for the BATS model, since there is only a small amount of autocorrelation left in the model's residuals. We conduct robustness checks with respect to the forecast horizon, the weekday used as a test set as well as the size of the training data. The surprising finding that the BATS model forecasts improve as the forecast horizon is increased might be due to the fact that the BATS model performs especially well on the weekdays following the Saturday used as the main test set. It seems like all forecasting models perform better on the weekdays and the test error raises for the weekend days. We hypothesize that the trip counts for the weekends contain more unpredictable fluctuations which might be correlated with major events in the city. The amount of training data seems to be conducive to the forecasting performance of the DLSTM model which might enable the algorithm to extract more information about the data.

Unexpected findings. To our surprise, the forecasting performance of the TBATS model with its trigonometric modeling procedure of multiple seasonal patterns diverges considerably form that of the BATS model. This finding is contradictory to the experiments of Livera et al. (2011), who find that the TBATS model performs superior for high-frequency data in 5 minute intervals. Furthermore, even though the BaggedETS ensemble model improves the baseline ETS model as expected, the obtained prediction intervals exhibit an exploding interval width. We assume that this finding is due to the fact that the MBB procedure cannot account for the weekly seasonal patterns present in hourly data. This can lead to exploding simulated sample paths, which results in unusually high upper quantiles of the simulated distribution.

Section 7

Conclusion

Using high frequency data from the New York City Open Data Smart City platform, we deploy the most recent ensemble methods and appropriate neural network model architectures to time series forecasting and compare these Machine Learning models to more traditional statistical models, which is the main focus of this paper. Consistent with the demands posed by the most recent M4 and M5 forecasting competitions, we also provide prediction intervals for all the models drawing on bootstrapping and quantile regression techniques. We also include statistical models which can account for multiple seasonal patterns explicitly and this feature seems to be crucial when it comes to forecasting performance. The final evaluation of traffic flow point forecasts shows that deep neural networks can considerably outperform all benchmark models and an LGBM boosting ensemble model of regression trees produces similarly accurate point forecasts as a BATS model that explicitly accounts for multiple seasonal periods. The BaggedETS ensemble model as well as the TBATS model surprisingly fall short of expectations since the former cannot outperform common benchmark models and the TBATS model's trigonometric seasonality modeling procedure does not seem to work well for forecasting high-frequency traffic flow data.

Quantile regression techniques seem to produce more accurate and narrow prediction intervals than taking quantiles of a boostrapped distribution or an assumed normal distribution. According to our robustness checks, the amount of training data seems to play the most important role in the forecasting performance. The weekday can influence the ranking of the test error of the best forecasting models considered in this paper.

Forecasting models which can account for multiple seasonal patterns can considerably outperform traditional statistical models which cannot model these characteristics observed for the high frequency data sets considered in this paper. We find that deep learning neural network models produce the most accurate point forecasts and minimizing a quantile regression loss can yield narrow prediction intervals with a high coverage probability. However, the deep neural networks require extensive computational resources and considerable technical expertise with respect to tuning the models. Exponential Smoothing state space models which can account for complex seasonal patterns, such as the BATS and the TBATS model, constitute a viable alternative with respect to the trade-off between forecasting performance and demands of computation and modeling.

Smart cities like New York City dispose of a large amount of high-frequency data which are particularly useful for the application of data intensive cutting edge Machine Learning models such as the boosting variants and recurrent neural networks. To our knowledge, there appears to be a gap between the prospects of such models and their adoption by smart city planners. The spectrum of possible application domains goes beyond the two use cases considered in this paper.

Appendix A

Formulae and State Space Equations for ETS Models

A.1 Recursive point forecast formulae

Trend	Seasonal		
	N	A	M
N	$\ell_t = \alpha y_t + (1-\alpha)\ell_{t-1}$ $\hat{y}_{t+h\|t} = \ell_t$	$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)\ell_{t-1}$ $s_t = \gamma(y_t - \ell_{t-1}) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t + s_{t-m+h_m^+}$	$\ell_t = \alpha(y_t / s_{t-m}) + (1-\alpha)\ell_{t-1}$ $s_t = \gamma(y_t / \ell_{t-1}) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t s_{t-m+h_m^+}$
A	$\ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1}$ $\hat{y}_{t+h\|t} = \ell_t + hb_t$	$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1}$ $s_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t + hb_t + s_{t-m+h_m^+}$	$\ell_t = \alpha(y_t / s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1}$ $s_t = \gamma(y_t / (\ell_{t-1} + b_{t-1})) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = (\ell_t + hb_t)s_{t-m+h_m^+}$
A_d	$\ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + \phi b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)\phi b_{t-1}$ $\hat{y}_{t+h\|t} = \ell_t + \phi_h b_t$	$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + \phi b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)\phi b_{t-1}$ $s_t = \gamma(y_t - \ell_{t-1} - \phi b_{t-1}) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t + \phi_h b_t + s_{t-m+h_m^+}$	$\ell_t = \alpha(y_t / s_{t-m}) + (1-\alpha)(\ell_{t-1} + \phi b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)\phi b_{t-1}$ $s_t = \gamma(y_t / (\ell_{t-1} + \phi b_{t-1})) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = (\ell_t + \phi_h b_t)s_{t-m+h_m^+}$
M	$\ell_t = \alpha y_t + (1-\alpha)\ell_{t-1}b_{t-1}$ $b_t = \beta^*(\ell_t / \ell_{t-1}) + (1-\beta^*)b_{t-1}$ $\hat{y}_{t+h\|t} = \ell_t b_t^h$	$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)\ell_{t-1}b_{t-1}$ $b_t = \beta^*(\ell_t / \ell_{t-1}) + (1-\beta^*)b_{t-1}$ $s_t = \gamma(y_t - \ell_{t-1}b_{t-1}) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t b_t^h + s_{t-m+h_m^+}$	$\ell_t = \alpha(y_t / s_{t-m}) + (1-\alpha)\ell_{t-1}b_{t-1}$ $b_t = \beta^*(\ell_t / \ell_{t-1}) + (1-\beta^*)b_{t-1}$ $s_t = \gamma(y_t / (\ell_{t-1}b_{t-1})) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t b_t^h s_{t-m+h_m^+}$
M_d	$\ell_t = \alpha y_t + (1-\alpha)\ell_{t-1}b_{t-1}^{\phi}$ $b_t = \beta^*(\ell_t / \ell_{t-1}) + (1-\beta^*)b_{t-1}^{\phi}$ $\hat{y}_{t+h\|t} = \ell_t b_t^{\phi_h}$	$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)\ell_{t-1}b_{t-1}^{\phi}$ $b_t = \beta^*(\ell_t / \ell_{t-1}) + (1-\beta^*)b_{t-1}^{\phi}$ $s_t = \gamma(y_t - \ell_{t-1}b_{t-1}^{\phi}) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t b_t^{\phi_h} + s_{t-m+h_m^+}$	$\ell_t = \alpha(y_t / s_{t-m}) + (1-\alpha)\ell_{t-1}b_{t-1}^{\phi}$ $b_t = \beta^*(\ell_t / \ell_{t-1}) + (1-\beta^*)b_{t-1}^{\phi}$ $s_t = \gamma(y_t / (\ell_{t-1}b_{t-1}^{\phi})) + (1-\gamma)s_{t-m}$ $\hat{y}_{t+h\|t} = \ell_t b_t^{\phi_h} s_{t-m+h_m^+}$

Figure A.1: Formulae for calculating point forecasts with ETS models

Notes: This figure contains the recursive formulae used to calculate point forecasts from ETS methods as given by R. J. Hyndman et al. (2008, p. 18). Each cell contains the forecast equation and the terms describing the components of the series that are used to apply the respective exponential smoothing method. β^* is defined as $\frac{\beta}{\alpha}$. h_m^+ is the number of remaining times in the forecast period up to and including time h, the forecast horizon. Therefore, h_m^+ can take on values $1, 2, \ldots, m$, where m is the number of periods in each season.

A.2 State space model equations for additive errors

Trend	Seasonal		
	N	A	M
N	$\mu_t = \ell_{t-1}$ $\ell_t = \ell_{t-1}(1+\alpha\varepsilon_t)$	$\mu_t = \ell_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \alpha(\ell_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = \ell_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1}(1+\alpha\varepsilon_t)$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
A	$\mu_t = \ell_{t-1} + b_{t-1}$ $\ell_t = (\ell_{t-1} + b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1})\varepsilon_t$	$\mu_t = \ell_{t-1} + b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = (\ell_{t-1} + b_{t-1})s_{t-m}$ $\ell_t = (\ell_{t-1} + b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1})\varepsilon_t$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
A_d	$\mu_t = \ell_{t-1} + \phi b_{t-1}$ $\ell_t = (\ell_{t-1} + \phi b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1})\varepsilon_t$	$\mu_t = \ell_{t-1} + \phi b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = (\ell_{t-1} + \phi b_{t-1})s_{t-m}$ $\ell_t = (\ell_{t-1} + \phi b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1})\varepsilon_t$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
M	$\mu_t = \ell_{t-1}b_{t-1}$ $\ell_t = \ell_{t-1}b_{t-1}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}(1+\beta\varepsilon_t)$	$\mu_t = \ell_{t-1}b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha(\ell_{t-1}b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = b_{t-1} + \beta(\ell_{t-1}b_{t-1} + s_{t-m})\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma(\ell_{t-1}b_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}(1+\beta\varepsilon_t)$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
M_d	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}^{\phi}(1+\beta\varepsilon_t)$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha(\ell_{t-1}b_{t-1}^{\phi} + s_{t-m})\varepsilon_t$ $b_t = b_{t-1}^{\phi} + \beta(\ell_{t-1}b_{t-1}^{\phi} + s_{t-m})\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma(\ell_{t-1}b_{t-1}^{\phi} + s_{t-m})\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}^{\phi}(1+\beta\varepsilon_t)$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$

Figure A.2: State Space Model Equations for ETS models with additive errors

Notes: This figure contains the state space model equations corresponding to the 15 methods from the classification of ETS models shown in Table 3.1 with additive error components as given by R. J. Hyndman et al. (2008, p. 21). l_t is the level term, b_t is the growth term and s_t denotes the seasonal term of the series at time t. The point forecast formulae can be rewritten in the forms of Equation (3.42) and Equation (3.43) by using the term components of the respective model as inputs for the state vector given in Equation (3.44).

A.3 State space model equations for multiplicative errors

Trend	Seasonal		
	N	A	M
N	$\mu_t = \ell_{t-1}$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t$	$\mu_t = \ell_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = \ell_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/\ell_{t-1}$
A	$\mu_t = \ell_{t-1} + b_{t-1}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t$	$\mu_t = \ell_{t-1} + b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = (\ell_{t-1} + b_{t-1})s_{t-m}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = b_{t-1} + \beta\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1} + b_{t-1})$
A_d	$\mu_t = \ell_{t-1} + \phi b_{t-1}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t$ $b_t = \phi b_{t-1} + \beta\varepsilon_t$	$\mu_t = \ell_{t-1} + \phi b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t$ $b_t = \phi b_{t-1} + \beta\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = (\ell_{t-1} + \phi b_{t-1})s_{t-m}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = \phi b_{t-1} + \beta\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1} + \phi b_{t-1})$
M	$\mu_t = \ell_{t-1}b_{t-1}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t/\ell_{t-1}$	$\mu_t = \ell_{t-1}b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = b_{t-1} + \beta\varepsilon_t/(s_{t-m}\ell_{t-1})$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1}b_{t-1})$
M_d	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha\varepsilon_t$ $b_t = b_{t-1}^{\phi} + \beta\varepsilon_t/\ell_{t-1}$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha\varepsilon_t$ $b_t = b_{t-1}^{\phi} + \beta\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha\varepsilon_t/s_{t-m}$ $b_t = b_{t-1}^{\phi} + \beta\varepsilon_t/(s_{t-m}\ell_{t-1})$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1}b_{t-1}^{\phi})$

Figure A.3: State Space Model Equations for ETS models with multiplicative errors

Notes: This figure contains the state space model equations corresponding to the 15 methods from the classification of ETS models shown in Table 3.1 with multiplicative error components as given by R. J. Hyndman et al. (2008, p. 22). l_t is the level term, b_t is the growth term and s_t denotes the seasonal term of the series at time t. The point forecast formulae can be rewritten in the forms of Equation (3.42) and Equation (3.43) by using the term components of the respective model as inputs for the state vector given in Equation (3.44). $\epsilon_t = \mu_t\epsilon_t$ is used for transformation from the point forecast formulae to the multiplicative state space model equations.

Appendix B

Supplementary results

B.1 Decompositions by the BATS and TBATS models

?? shows the decomposition of the training data obtained from the BATS(0.035, 4,1, 0.934, 24,168) model. **??** shows the decomposition of the training data obtained from the TBATS(1, 5,1, -, <24,8>,<168,5>) model.

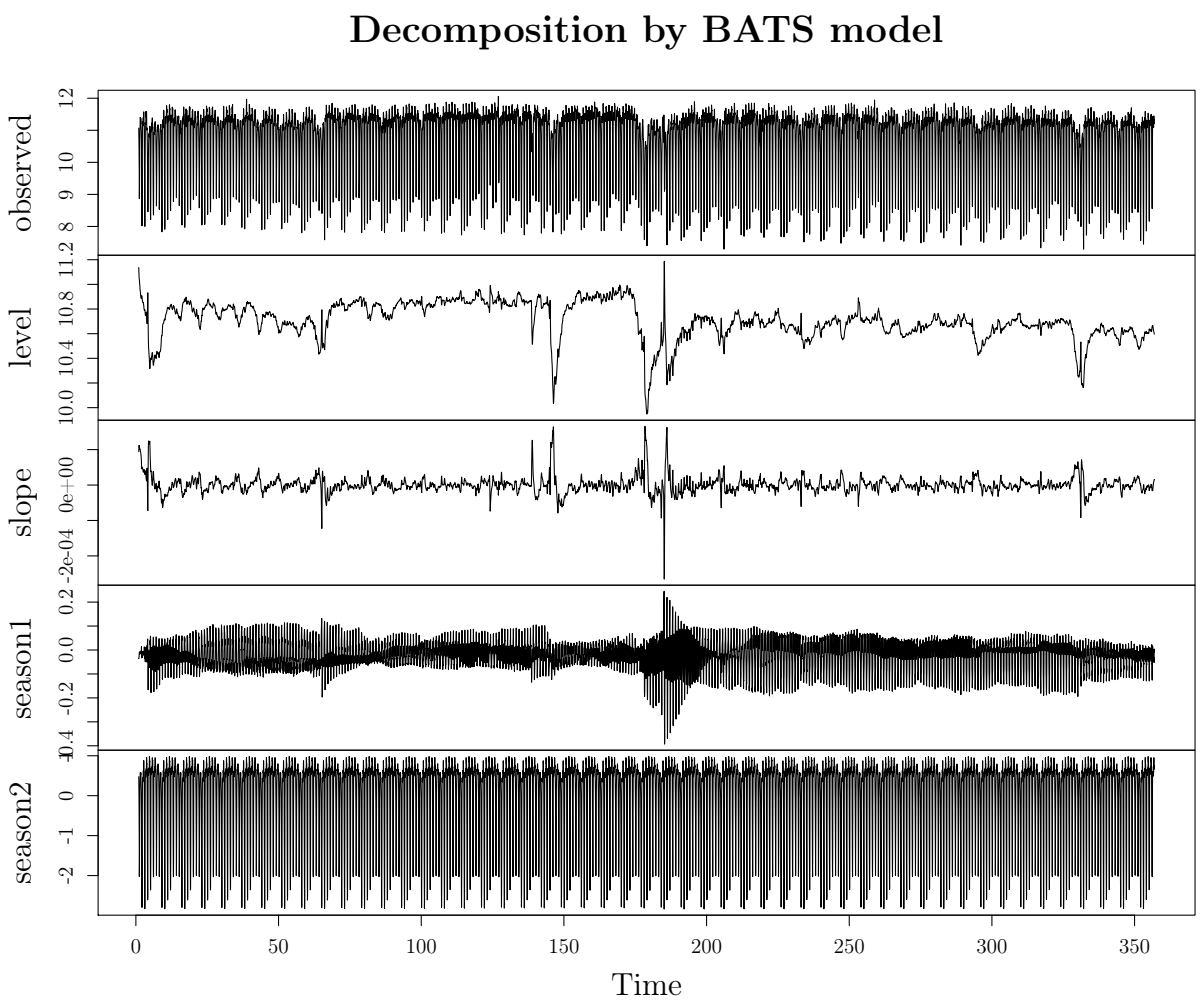

Figure B.1: Trigonometric decomposition of the traffic flow time series through BATS

Notes: This figure shows the decomposition of the observed traffic flow time series shown in the panel "observed" into the trend component of the series shown in the "level" panel, as well as the three seasonal patterns. Panel "season1" shows the weekly seasonality component and the panel "season2" shows the 24 hourly or daily seasonality.

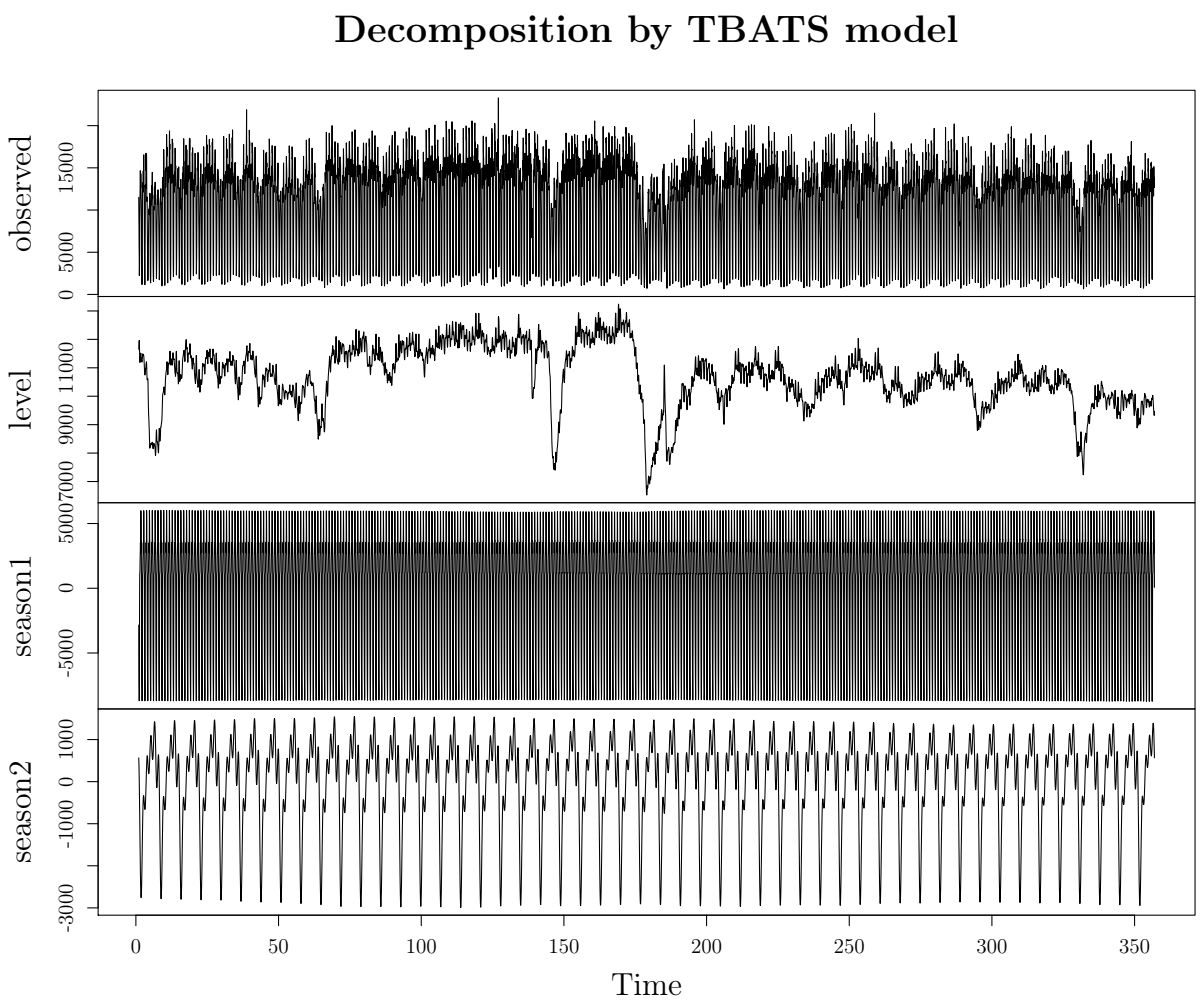

Figure B.2: Trigonometric decomposition of the traffic flow time series through TBATS

Notes: This figure shows the decomposition of the observed traffic flow time series shown in the panel "observed" into the trend component of the series shown in the "level" panel, as well as the three seasonal patterns. Panel "season1" shows the weekly seasonality component and the panel "season2" shows the 24 hourly or daily seasonality.

B.2 Gridsearch results for Deep neural networks

Table B.1: Model setup for the DGRU and the DLSTM neural networks (traffic data)

Model Setup component	**Component details**	
Prediction target	24 hours ahead traffic flows : $y_{T+1}, \ldots, y_{T+24}$	
Input variables	History of 60 lags : $\{y_T, y_{T-1}, \ldots, y_{T-59}\}$	
Output variables	Predictions of 24 steps ahead: $\{\hat{y}_{T+1}, \hat{y}_{t+2}, \ldots, \hat{y}_{T+24}\}$	
Optimal training hyperparameters		
	DGRU	**DLSTM**
Activation function	'Leaky Relu'	'Leaky Relu'
Learning rate	0.001	0.001
Number of neurons	1000	1500
Batch size	100	100
Number of epochs	250	250
Optimizer	'Adadelta'	'Adagrad'
Dropout	0.0	0.0

Notes: This table contains all information about the model setup used for both the DGRU and the DLSTM neural networks. The training hyperparameters are the ones obtained from the respective grid search cross validation procedures which are carried out for each RNN architecture type separately. Both stacked hidden layers for each of the neural network types are identical.

References

Ahmed, N. K., Atiya, A. F., Gayar, N. E., & El-Shishiny, H. (2010). An empirical comparison of machine learning models for time series forecasting. *Econometric Reviews*, *29*(5-6), 594–621. doi:https://doi.org/10.1080/07474938.2010.481556

Amini, A., & Soleimany, A. (2020a). *MIT 6.S191: Introduction to Deep Learning - Lecture 1: Intro to Deep Learning*. [University Lecture Notes]. Massachusetts Institute of Technology. Retrieved from http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf

Amini, A., & Soleimany, A. (2020b). *MIT 6.S191: Introduction to Deep Learning - Lecture 2: Deep Sequence Modeling*. [University Lecture Notes]. Massachusetts Institute of Technology. Retrieved from http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L2.pdf

Aringhieri, R., Bruni, M., Khodaparasti, S., & van Essen, J. (2017). Emergency medical services and beyond: Addressing new challenges through a wide literature review. *Computers and Operations Research*, *78*, 349–368. doi:https://doi.org/10.1016/j.cor.2016.09.016

Armstrong, J. S. (2001). *Principles of forecasting: A handbook for researchers and practitioners*. doi:https://doi.org/10.1007/978-0-306-47630-3

Baker, J., & Fitzpatrick, K. (1986). Determination of an optimal forecast model for ambulance demand using goal programming. *Journal of the Operational Research Society*, *37*(11), 1047–1059. doi:https://doi.org/10.1057/jors.1986.182

Bandara, K., Shi, P., Bergmeir, C., Hewamalage, H., Tran, Q., & Seaman, B. (2019). Sales demand forecast in e-commerce using a long short-term memory neural network methodology. In *International conference on neural information processing* (pp. 462–474). Springer. doi:https://doi.org/10.1007/978-3-030-36718-3

Barrow, D. K., & Crone, S. F. (2016). A comparison of adaboost algorithms for time series forecast combination. *International Journal of Forecasting*, *32*(4), 1103–1119. doi:https://doi.org/10.1016/j.ijforecast.2016.01.006

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade* (pp. 437–478). Springer. Retrieved from https://arxiv.org/pdf/1206.5533v2.pdf

Bengio, Y. et al. (2009). Learning deep architectures for ai. *Foundations and trends in Machine Learning*, *2*(1), 1–127. doi:http://dx.doi.org/10.1561/2200000006

BenTaieb, S., Bontempi, G., Atiya, A., & Sorjamaa, A. (2012). A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition. *Expert Systems with Applications*, *39*(8), 7067–7083. doi:https://doi.org/10.1016/j.eswa.2012.01.039

Bergmeir, C., & Benítez, J. M. (2012). On the use of cross-validation for time series predictor evaluation. *Information Sciences*, *191*, 192–213. Data Mining for Software Trustworthiness. doi:https://doi.org/10.1016/j.ins.2011.12.028

Bergmeir, C., Hyndman, R. J., & Koo, B. (2018). A note on the validity of cross-validation for evaluating autoregressive time series prediction. *Computational Statistics & Data Analysis*, *120*, 70–83. doi:https://doi.org/10.1016/j.csda.2017.11.003

Bergmeir, C., Hyndman, R. J., & Benıtez, J. M. (2016). Bagging exponential smoothing methods using stl decomposition and box–cox transformation. *International journal of forecasting*, *32*(2), 303–312. doi:https://doi.org/10.1016/j.ijforecast.2015.07.002

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, *13*(1), 281–305. Retrieved from https://dl.acm.org/doi/10.5555/2188385.2188395

Best, H., & Wolf, C. (2014). *The sage handbook of regression analysis and causal inference*. Sage. Retrieved from https://dx.doi.org/10.4135/9781446288146

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer. Retrieved from https://dl.acm.org/doi/book/10.5555/1162264

Box, G. E., & Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society: Series B (Methodological)*, *26*(2), 211–243. doi:https://doi.org/10.1111/j.2517-6161.1964.tb00553.x

Box, G. E., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). *Time series analysis: Forecasting and control*. John Wiley & Sons.

Breiman, L. (1996). Bagging predictors. *Machine learning*, *24*(2), 123–140. doi:https://doi.org/10.1023/A:1018054314350

Breiman, L. et al. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, *16*(3), 199–231. Retrieved from http://www.jstor.org/stable/2676681

Brockwell, P. J., & Davis, R. A. (2016). *Introduction to time series and forecasting*. doi:https://doi.org/10.1007/978-3-319-29854-2

Brown, R. G. (1959). *Statistical forecasting for inventory control*. McGraw/Hill.

Brownlee, J. (2017). *Long short-term memory networks with python: Develop sequence prediction models with deep learning*. Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/lstms-with-python/

Brownlee, J. (2018). *Better deep learning: Train faster, reduce overfitting, and make better predictions*. Machine Learning Mastery. Retrieved from https://books.google.dk/books?id=T1-nDwAAQBAJ

Channouf, N., L'Ecuyer, P., Ingolfsson, A., & Avramidis, A. N. (2007). The application of forecasting techniques to modeling emergency medical system calls in calgary, alberta. *Health care management science*, *10*(1), 25–45. doi:https://doi.org/10.1007/s10729-006-9006-3

Chatfield, C. (2001). Prediction intervals for time-series forecasting. In J. S. Armstrong (Ed.), *Principles of forecasting: A handbook for researchers and practitioners* (pp. 475–494). doi:10.1007/978-0-306-47630-3_21

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 785–794). doi:https://doi.org/10.1145/2939672.2939785

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Conference on empirical methods in natural language processing (emnlp 2014)*. doi:https://doi.org/10.3115/v1/D14-1179

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., & LeCun, Y. (2015). The Loss Surfaces of Multilayer Networks. In G. Lebanon & S. V. N. Vishwanathan (Eds.), *Proceedings of the eighteenth international conference on artificial intelligence and statistics* (Vol. 38, pp. 192–204). Proceedings of Machine Learning Research. San Diego, California, USA: PMLR. Retrieved from http://proceedings.mlr.press/v38/choromanska15.html

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Nips 2014 workshop on deep learning, december 2014*. Retrieved from https://arxiv.org/abs/1412.3555v1

Cromwell, J. B., Labys, W. C., & Terraza, M. (1994). *Univariate tests for time series models*. doi:https://dx.doi.org/10.4135/9781412986458

De Oliveira, E. M., & Cyrino Oliveira, F. L. (2018). Forecasting mid-long term electric energy consumption through bagging arima and exponential smoothing methods. *Energy*, *144*, 776–788. doi:https://doi.org/10.1016/j.energy.2017.12.049

Deng, H., Runger, G., Tuv, E., & Vladimir, M. (2013). A time series forest for classification and feature extraction. *Information Sciences*, *239*, 142–153. doi:https://doi.org/10.1016/j.ins.2013.02.030

Enders, W. (2014). *Applied econometric times series*. Wiley Series in Probability and Statistics. Wiley. Retrieved from https://www.wiley.com/en-us/Applied+Econometric+Time+Series%2C+4th+Edition-p-9781118918623

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational statistics and data analysis*, *38*(4), 367–378. doi:https://doi.org/10.1016/S0167-9473(01)00065-2

Fu, R., Zhang, Z., & Li, L. (2016). Using lstm and gru neural network methods for traffic flow prediction. In *2016 31st youth academic annual conference of chinese association of automation (yac)* (pp. 324–328). IEEE. doi:https://doi.org/10.1109/YAC.2016.7804912

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. Retrieved from http://www.deeplearningbook.org

Hamilton, J. D. (1994). *Time series analysis*. Princeton New Jersey. Retrieved from https://press.princeton.edu/books/ebook/9780691218632/time-series-analysis

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction*. doi:https:||doi.org/10.1007/978-0-387-84858-7

Haykin, S. (1999). *Neural networks: A comprehensive foundation*. 2nd edition. Prentice Hall.

Hermans, M., & Schrauwen, B. (2013). Training and analysing deep recurrent neural networks. In *Proceedings of the 26th international conference on neural information processing systems - volume 1* (pp. 190–198). Retrieved from https://dl.acm.org/doi/10.5555/2999611.2999633

Hewamalage, H., Bergmeir, C., & Bandara, K. (2019). Recurrent neural networks for time series forecasting: Current status and future directions. *arXiv preprint arXiv:1909.00590*. Retrieved from https://arxiv.org/abs/1909.00590

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. doi:https://doi.org/10.1162/neco.1997.9.8.1735

Hyndman, R. J., & Athanasopoulos, G. (2018). *Forecasting: Principles and practice*. OTexts. Retrieved from https://otexts.com/fpp2/

Hyndman, R. J., & Khandakar, Y. (2008). Automatic time series forecasting: The forecast package for r 7, 2008. *Journal of Statistical Software*, *27*. doi:https://dx.doi.org/10.18637/jss.v027.i03

Hyndman, R. J., & Koehler, A. B. (2006). Another look at measures of forecast accuracy. *International journal of forecasting*, *22*(4), 679–688. doi:https://doi.org/10.1016/j.ijforecast.2006.03.001

Hyndman, R. J., Koehler, A. B., Ord, J. K., & Snyder, R. D. (2008). *Forecasting with exponential smoothing: The state space approach*. doi:https://doi.org/10.1007/978-3-540-71918-2

Hyndman, R. J., Koehler, A. B., Snyder, R. D., & Grose, S. (2002). A state space framework for automatic forecasting using exponential smoothing methods. *International Journal of Forecasting*, *18*(3), 439–454. doi:https://doi.org/10.1016/S0169-2070(01)00110-8

Ismagilova, E., Hughes, L., Dwivedi, Y. K., & Raman, K. R. (2019). Smart cities: Advances in research—an information systems perspective. *International Journal of Information Management*, *47*, 88–100. doi:https://doi.org/10.1016/j.ijinfomgt.2019.01.004

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning*. doi:https://doi.org/10.1007/978-1-4614-7138-7

Januschowski, T., Gasthaus, J., Wang, Y., Salinas, D., Flunkert, V., Bohlke-Schneider, M., & Callot, L. (2020). Criteria for classifying forecasting methods. *International Journal of Forecasting*, *36*(1), 167–177. doi:https://doi.org/10.1016/j.ijforecast.2019.05.008

Jebb, A. T., Tay, L., Wang, W., & Huang, Q. (2015). Time series analysis for psychological research: Examining and forecasting change. *Frontiers in psychology*, *6*, 727. doi:https://doi.org/10.3389/fpsyg.2015.00727

John, R., & Townshend, L. (2019). *CS 229: Machine Learning - Section 7: Decision Trees*. [University Lecture Notes]. Stanford University. Retrieved from http://cs229.stanford.edu/notes/cs229-notes-dt.pdf

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems* (pp. 3146–3154). Retrieved from http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

Khosravi, A., Nahavandi, S., Creighton, D., & Atiya, A. F. (2010). Lower upper bound estimation method for construction of neural network-based prediction intervals. *IEEE transactions on neural networks*, *22*(3), 337–346. doi:https://doi.org/10.1109/TNN.2010.2096824

Khosravi, A., Nahavandi, S., Creighton, D., & Atiya, A. F. (2011). Comprehensive review of neural network-based prediction intervals and new advances. *IEEE Transactions on neural networks*, *22*(9), 1341–1356. doi:https://doi.org/10.1109/TNN.2011.2162110

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. Retrieved from https://arxiv.org/pdf/1412.6980.pdf

Künsch, H. R. (1989). The jackknife and the bootstrap for general stationary observations. *The annals of Statistics*, 1217–1241. doi:https://doi.org/10.1214/aos/1176347265

Liu, J., Wu, C., & Wang, J. (2018). Gated recurrent units based neural network for time heterogeneous feedback recommendation. *Information Sciences*, *423*, 50–65. doi:https://doi.org/10.1016/j.ins.2017.09.048

Livera, A. M. D., Hyndman, R. J., & Snyder, R. D. (2011). Forecasting time series with complex seasonal patterns using exponential smoothing. *Journal of the American Statistical Association*, *106*(496), 1513–1527. doi:https://doi.org/10.1198/jasa.2011.tm09771

Livni, R., Shalev-Shwartz, S., & Shamir, O. (2014). On the computational efficiency of training neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 27* (pp. 855–863).

Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/5267-on-the-computational-efficiency-of-training-neural-networks.pdf

Ma, X., Tao, Z., Wang, Y., Yu, H., & Wang, Y. (2015). Long short-term memory neural network for traffic speed prediction using remote microwave sensor data. *Transportation Research Part C: Emerging Technologies*, *54*, 187–197. doi:https://doi.org/10.1016/j.trc.2015.03.014

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018). Statistical and machine learning forecasting methods: Concerns and ways forward. *PloS one*, *13*(3), e0194889. doi:https://doi.org/10.1371/journal.pone.0194889

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2020). The m4 competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, *36*(1), 54–74. M4 Competition. doi:https://doi.org/10.1016/j.ijforecast.2019.04.014

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133. doi:https://doi.org/10.1007/BF02478259

Meijer, A., & Bolívar, M. P. R. (2016). Governing the smart city: A review of the literature on smart urban governance. *International Review of Administrative Sciences*, *82*(2), 392–408. doi:https://doi.org/10.1177/0020852314564308

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. MIT press. Retrieved from https://mitpress.mit.edu/books/machine-learning-1

Nagy, A. M., & Simon, V. (2018). Survey on traffic prediction in smart cities. *Pervasive and Mobile Computing*, *50*, 148–163. doi:https://doi.org/10.1016/j.pmcj.2018.07.004

Ng, A. (2019). *CS 229: Machine Learning - Lecture 2: Supervised Learning*. [University Lecture Notes]. Stanford University. Retrieved from http://cs229.stanford.edu/notes2019fall/cs229-notes1.pdf

Pan, L., & Politis, D. N. (2016). Bootstrap prediction intervals for linear, nonlinear and nonparametric autoregressions. *Journal of Statistical Planning and Inference*, *177*, 1–27. doi:https://doi.org/10.1016/j.jspi.2014.10.003

Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y. (2013). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*. Retrieved from https://arxiv.org/abs/1312.6026

Petropoulos, F., Hyndman, R. J., & Bergmeir, C. (2018). Exploring the sources of uncertainty: Why does bagging for time series forecasting work? *European Journal of Operational Research*, *268*(2), 545–554. doi:https://doi.org/10.1016/j.ejor.2018.01.045

Rojas, R. (1996). *Neural networks: A systematic introduction*. doi:https://doi.org/10.1007/978-3-642-61068-4

Romano, Y., Patterson, E., & Candes, E. (2019). Conformalized quantile regression. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 3543–3553). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/8613-conformalized-quantile-regression.pdf

Ruder, S. (2017). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*. Retrieved from https://arxiv.org/pdf/1609.04747.pdf

Safikhani, A., Kamga, C., Mudigonda, S., Faghih, S. S., & Moghimi, B. (2020). Spatio-temporal modeling of yellow taxi demands in new york city using generalized star models. *International Journal of Forecasting*, *36*(3), 1138–1148. doi:https://doi.org/10.1016/j.ijforecast.2018.10.001

Setzler, H., Saydam, C., & Park, S. (2009). Ems call volume predictions: A comparative study. *Computers and Operations Research*, *36*(6), 1843–1851. doi:https://doi.org/10.1016/j.cor.2008.05.010

Shumway, R. H., & Stoffer, D. S. (2017). *Time series analysis and its applications: With r examples*. doi:https://doi.org/10.1007/978-3-319-52452-8

Smyl, S. (2020). A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. *International Journal of Forecasting*, *36*(1), 75–85. M4 Competition. doi:https://doi.org/10.1016/j.ijforecast.2019.03.017

Sussillo, D., & Barak, O. (2013). Opening the black box: Low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, *25*(3), 626–649. doi:https://doi.org/10.1162/NECO_a_00409

Taieb, S. B., & Hyndman, R. (2014). Boosting multi-step autoregressive forecasts. In *International conference on machine learning* (pp. 109–117). Retrieved from https://dl.acm.org/doi/abs/10.5555/3044805.3044819

Tascikaraoglu, A. (2018). Evaluation of spatio-temporal forecasting methods in various smart city applications. *Renewable and Sustainable Energy Reviews*, *82*, 424–435. doi:https://doi.org/10.1016/j.rser.2017.09.078

Vlahogianni, E. I., Golias, J. C., & Karlaftis, M. G. (2004). Short-term traffic forecasting: Overview of objectives and methods. *Transport reviews*, *24*(5), 533–557. doi:https://doi.org/10.1080/0144164042000195072

Vlahogianni, E. I., Karlaftis, M. G., & Golias, J. C. (2014). Short-term traffic forecasting: Where we are and where we're going. *Transportation Research Part C: Emerging Technologies*, *43*, 3–19. Special Issue on Short-term Traffic Flow Forecasting. doi:https://doi.org/10.1016/j.trc.2014.01.005

West, M., & Harrison, J. (2006). *Bayesian forecasting and dynamic models*. doi:https://doi.org/10.1007/b98971

Yan, W. (2012). Toward automatic time-series forecasting using neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, *23*(7), 1028–1039. doi:https://doi.org/10.1109/TNNLS.2012.2198074

Zhao, Z., Chen, W., Wu, X., Chen, P. C., & Liu, J. (2017). Lstm network: A deep learning approach for short-term traffic forecast. *IET Intelligent Transport Systems*, *11*(2), 68–75. doi:https://doi.org/10.1049/iet-its.2016.0208