Evaluation of Deep Learning Approaches for Predicting Electric Vehicle Energy Demand and Charging Station Occupancy

Masterarbeit

vorgelegt von

Dr. med. Matthias Misiewicz

zur Erlangung des akademischen Grades

Master of Science Scientific Computing

Gutachter:

Prof. Dr. Gabriele Steidl

Dr. Serafin von Roon

Technische Universität Berlin

Fakultät für Mathematik und Naturwissenschaften

Institut für Mathematik

Berlin, den 22. März 2023

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 22. März 2023, Matthias Misiewicz

Zusammenfassung

Die genaue Vorhersage des Energiebedarfs von Elektrofahrzeugen (EV) ist von großer Bedeutung für den effizienten und sicheren Betrieb von EV-Ladestationen und Stromnetzen, während die damit verbundene Belegungsprognose von EV-Ladestationen es EV-Nutzern ermöglicht, Ladevorgänge besser zu planen und somit die breite Akzeptanz der Elektromobilität fördert. Diese Aufgaben sind Zeitreihenprognoseprobleme, für die Deep-Learning-Modelle in jüngster Zeit erhebliches Potenzial gezeigt haben. Diese Arbeit bewertet die Leistung von Deep-Learning-Modellen bei der Prognose des aggregierten Energiebedarfs von EV-Ladestationen und der Vorhersage der Belegung einzelner Stationen. Eine Literaturrecherche wird durchgeführt, um geeignete Deep-Learning-Architekturen zu identifizieren. Die ausgewählten Modelle umfassen vier rekurrente Netze, ein eindimensionales Convolutional Neural Network (1D CNN), zwei Single-Stream-Hybridnetze mit rekurrenten und konvolutionellen Elementen sowie zwei Multi-Stream-Hybridnetze, die rekurrente Komponenten mit konvolutionellen oder dichten Komponenten kombinieren. Die neuronalen Netze werden mit AdaBoost und XGBoost verglichen, die wesentlich einfacher zu implementieren sind, während Lineare Regression und Logistische Regression als Basismodelle dienen. Nach der Etablierung eines theoretischen Rahmens werden die Modelle anhand heterogener Datensätze bewertet, die einzelne Ladestationen für die Belegungsprognose und aggregierte Zeitreihen für die Energiebedarfsprognose repräsentieren. Die Hyperparameteroptimierung erfolgt mit Hilfe eines Random-Search-Ansatzes auf einem Hold-Out-Validierungsdatensatz. Die Leistung der Modelle wird über drei Horizonte hinweg bewertet. Die Ergebnisse zeigen, dass das CNN Long Short-Term Memory (CNN-LSTM) in der Energiebedarfsprognose am besten abschneidet, und das Multi-Stream-Netz "Hybrid LSTM" die insgesamt beste Leistung für die Belegungsprognose aufweist. Allerdings könnte eine Fehlerakkumulation bei der rekursiven Energiebedarfsprognose die Bewertung der Benchmark-Modelle beeinflussen.

Abstract

The accurate prediction of electric vehicle (EV) energy demand is of great significance for the efficient and secure operation of EV charging stations and power grids, while the related task of EV charging station occupancy prediction enables EV users to better plan their charging processes, thereby fostering the widespread adoption of electromobility. These tasks are time series forecasting problems, for which deep learning models have recently shown considerable potential. This work evaluates the performance of deep learning models in forecasting aggregated EV charging station energy demand and predicting occupancy of individual stations. A literature review is conducted to identify suitable deep learning architectures. The chosen models include four recurrent networks, a one-dimensional convolutional neural network (1D CNN), two single-stream hybrid architectures with recurrent and convolutional components, and two multi-stream hybrid architectures, each combining recurrent components with either convolutional or fully connected components for energy demand and occupancy prediction, respectively. The neural networks are benchmarked against AdaBoost and XGBoost, which are fundamentally easier to implement, while Linear Regression and Logistic Regression serve as respective baseline models. After establishing a theoretical framework, the models are evaluated using heterogenous datasets representing single charging stations for occupancy forecasting and aggregated time series for energy demand forecasting. Hyperparameter optimization is conducted using a random search approach on a hold-out validation set. The performance of the models is assessed across three forecasting horizons. The results suggest that CNN Long Short-Term Memory (CNN-LSTM) outperforms its counterparts in EV energy demand forecasting, whereas the multi-stream network, termed "Hybrid LSTM", exhibits the most favorable performance for occupancy state prediction. However, possible error accumulation in recursive forecasting in the energy demand scenario may affect the performance evaluation of the benchmark models.

Acknowledgements

I would like to express my gratitude to Prof. Dr. Gabriele Steidl, Adrian Ostermann, and Theodor Haug for their support and willingness to share their time and expertise with me and to provide feedback on my work. I would also like to extend my gratitude to my parents for their support and encouragement, which has been a source of strength and motivation for me.

Contents

Chapter 1

Introduction

Global warming caused by greenhouse gas (GHG) emissions is considered one of the major challenges of the 21st century. As of 2018, more than 12% of GHG emissions in the EU were caused directly by passenger cars. The transport sector saw a net increase of said emissions between 2013 and 2019 [1].

The European Union aims to achieve climate neutrality by 2050, a goal that can only be reached by the massive adoption of new technologies. An increase in energy efficiency, far-reaching electrification in the sectors of industry, buildings and transport as well as exclusively climate-neutral electricity generation are necessary to reduce GHG emissions appropriately [2, 3].

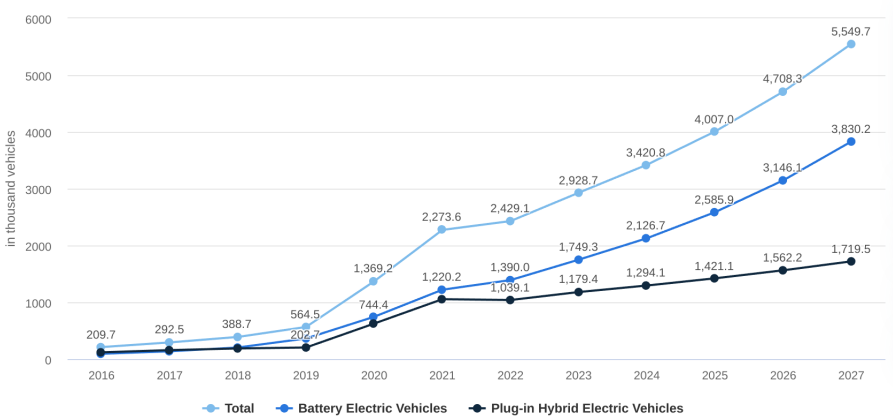


Figure 1.1: Projected EV sales in Europe [4].

The EU aims at a reduction of transport sector-related emissions by 60% compared to 1990. One highly promising to reduce transport sector emissions is the large-scale adoption of electric vehicles (EVs). Urban air pollution and its effect on health are additional issues that have prompted politicians to advocate for the use of EVs as a

substitute for traditional gasoline-powered vehicles [5]. For example, the German government plans to reach 10 Million EVs and one Million public EV charging stations by 2030 [6], which aligns with the European Alternative Fuels Infrastructure Directive [7].

Aside from the high acquisition costs of EVs, the main obstacles to purchasing a EV are insufficiency and unavailability of charging infrastructure [8, 9]. Only a fraction of charging stations in the EU is fast-charging, causing typical charging sessions to take several hours [10]. For EV users, these deficiencies are associated with complicated route planning, range anxiety [11, 12, 13] and possible long waiting times at charging stations.

Despite these limitations, fully battery electric vehicles accounted for 12.1% of car sales in EU markets in 2022, compared to just 1.9% in 2019 [14]. Due to strong political support, increasing market penetration of EVs can be expected in the future.

1.1 Motivation

The widespread adoption of electric vehicles presents challenges and opportunities for the existing power system. Unlike normal industrial or household loads, EV loads are unique in both periodicity and fluctuation [15]. The increased demand from EVs could place a significant additional load on the network, potentially compromising its stability, particularly in low-voltage networks where the grid is not equipped to handle the extra demand. Moreover, forecasting EV energy demand across geographically distributed areas covering whole cities is essential for grid stability, infrastructure planning, renewable energy integration, and demand response management [16]. This type of forecasting allows grid operators to have a comprehensive understanding of the energy demand across the entire region, enabling them to make better decisions about resource allocation and infrastructure investments. It also complements charging station energy demand forecasting by providing a macro-level perspective that can help identify potential bottlenecks, optimize the distribution network, and facilitate the integration of renewable energy sources [17].

EV energy demand forecasting aims to predict the amount of electricity required by an EV charging site, which typically consists of one or more individual charging stations, at a certain time. On the other hand, EV charging station occupancy forecasting involves predicting the usage of individual charging stations by EV users at a given time. These two forecasting scenarios are related as charging station occupancy affects the energy demand. If charging station occupancy is high, the energy demand will also be

high, requiring the generation and distribution of more electricity. Both forecasting strategies can enable grid operators to better plan and manage the generation and distribution of electricity, thereby circumventing overloads and blackouts [18]. They can help optimize the utilization of resources and facilitate the integration of renewable energy sources. In addition, precise occupancy prediction of individual EV charging stations helps charging station operators optimize charging schedules, balance the load and avoid congestion. EV users benefit from accurate occupancy prediction as it reduces waiting times, makes trip planning convenient and can extenuate range anxiety [19].

The intermittent and highly stochastic charging behavior of EV users in time and space makes both EV energy demand forecasting and charging station occupancy prediction with satisfactory accuracy an increasingly difficult task that traditional statistical forecasting methods are unable to fulfill, which calls for more powerful prediction models [20]. Further, several factors such as weather, holidays, real-time electricity prices, and human behavior have an increasing impact on the currently diversifying power grid systems and exponentially growing amounts of complex data are captured [21].

The previously mentioned aspects indicate the need for forecasting models with robust adaptive learning and generalization abilities that can leverage the availability of large amounts of high-dimensional data. Numerous deep learning methods meet these demands and have become successful in both energy demand and occupancy forecasting tasks.

1.2 Objective and Scope

This thesis aims to identify deep learning architectures that are suitable for accurate EV energy demand prediction and charging station occupancy prediction. Specifically, we want to examine if there exist architectures that consistently demonstrate superior predictive capabilities across multiple heterogeneous datasets.

To achieve this, several candidate architectures are selected based on a literature review, with the goal of finding architectures suitable for both energy demand and occupancy prediction tasks, thereby minimizing the implementation overhead within this study. The predictive performance of the chosen architectures is subsequently assessed, following two distinct approaches:

In the context of the energy demand prediction, the predictive performance of the selected models is evaluated on four time series datasets. Two of these datasets represent aggregated EV charging station energy demand from specific charging sites with a considerable number of charging stations, exhibiting regular user patterns. The remaining two time series represent the energy demand of EV charging stations spread across larger geographic ranges, covering areas of entire cities. These datasets showcase more irregular patterns due to the overall less synchronized user behavior.

In the occupancy scenario, we use the same source data; however, the time series are not aggregated, resulting in a large number of individual series representing occupancy state profiles of individual charging stations. Again, due to larger geographic ranges in two of the original datasets, we anticipate an overall higher presence of irregular patterns in the corresponding time series, while the series originating from distinct charging sites are expected to exhibit more synchronized user behavior.

In summary, this thesis aims to provide stakeholders with valuable insights that facilitate the identification of the most suitable deep learning architectures for the respective forecasting tasks, irrespective of the underlying user patterns.

1.3 Thesis Outline

This work is organized into five main chapters.

- **Related Work** - Assessment of the current state of research, exploring which models are commonly used for the tasks of energy demand and charging station occupancy forecasting.
- **Theory** - Presentation of a theoretical foundation for deep learning and time series problems, including theoretical aspects of the evaluated deep learning models, baseline models, benchmark models, and evaluation metrics for time series forecasting.
- **Methods** - Explanation of the experimental methodology used, covering data preparation, time series generation, feature selection, data processing, and hyperparameter optimization.
- **Results** - Presentation and discussion of the results obtained during the experiments.
- **Conclusions** - Final assessment of model performance for each task, including limitations of this study and recommendations for future work.

Chapter 2

Related Work

Models for performing energy demand forecasting, which represents a regression problem, can be categorized into statistical and artificial intelligence models. Traditional models include the Autoregressive (AR) method, the Moving Average (MA) method, combined approaches such as Autoregressive Moving Average (ARMA), Autoregressive Integrated Moving Average (ARIMA), advanced combined approaches such as Seasonal Autoregressive Integrated Moving Average with Exogenous Regressors (SARIMAX), methods based on exponential smoothing such as Simple Exponential Smoothing (SES) and Holt Winter's Exponential Smoothing (HWES) [22]. These traditional methods for time series forecasting are based on the Box-Jenkins methodology. They have been used for many decades for predicting the future in economics, science, industry and finance. However, they exhibit certain weaknesses when it comes to real-world scenarios: They rely on strong assumptions about the series, such as stationarity and linearity, cannot handle missing or corrupt data, assume a fixed temporal dependence between observations and focus only on one-step-forecasts [23, 24]. Their use is limited to settings where the aforementioned assumptions hold and a-priori-knowledge of the underlying system is available [25]. Furthermore, selecting model parameters and time series features demands skill and expertise [22]. To perform forecasting with traditional methods, a given time series must be thoroughly analyzed and transformed if necessary, which can be an elaborate process [26].

On the other hand, deep neural networks can model complex nonlinear relationships and are suitable for multiple input and output scenarios. They profit from large amounts of data and can learn features automatically during training, thus alleviating the need for elaborate time series analysis and feature selection [27, 28]. Recently, both academia and industry have shown a growing interest in deep neural networks. This rise in popularity can be attributed to several factors [29, 30]:

- Increased availability of large training datasets with high-quality labels
- Advances in hardware performance and GPU-based parallel computing
- Highly specialized and user-friendly software frameworks such as PyTorch, Tensorflow, Caffe, Keras, BigDl, etc.
- Improved regularization techniques.

In several recent studies, deep neural networks have shown promising results in the domain of energy demand forecasting at varying forecasting horizons [31, 32, 33, 34]. Other machine learning algorithms that were used successfully for energy demand forecasting at multiple forecasting horizons are Random Forest (RF) [35, 36], Support Vector Regression (SVR) [37, 38], K-Nearest Neighbors Regression (KNN) [39, 40], as well as the boosting ensemble methods AdaBoost [41, 42] and XGBoost [43, 44].

The following findings all fall within the scope of general energy demand forecasting. The authors in [45] found superior predictive accuracy when comparing a deep recurrent neural network (RNN) with a shallow neural network. In [46], the authors compared a hybrid architecture consisting of a Convolutional Neural Network (CNN) module and a Long Short-Term Memory (LSTM) module to RF, LSTM and CNN. They found a superior performance of the hybrid model. A parallel CNN-RNN outperformed SVR and DFN (deep feedforward neural network) in one-day ahead forecasting in [47]. The authors in [48] proposed a parallel CNN-LSTM architecture that outperformed ARIMA, SES, SVR, DFN and LSTM in short-term forecasting on two datasets. XGBoost exhibited a higher predictive accuracy than SVR, LSTM and a deep neural network based on Gated Recurrent Units (GRU)[49]. Similarly, XGBoost outperformed LSTM at a forecasting horizon of 24 hours in [50]. In [42], it was found that AdaBoost based on linear regressors can have higher predictive accuracy than XGBoost, RF and SVR at a medium-term forecasting horizon.

The authors in [51] compared DFN, RNN, LSTM and GRU for forecasting short-term energy demand at a large public EV charging site in Shenzhen and attributed the strongest predictive performance to the GRU model. In [30], the same dataset was used to evaluate the predictive performance of an LSTM network and a non-recurrent deep neural network at two time scales, with the LSTM model showing superior results.

Occupancy state prediction of individual EV charging stations is commonly treated as a classification problem. Traditional statistical forecasting methods are of limited use for this scenario due to the significant impact of random fluctuations on the oc-

cupancy state profiles of individual charging stations [52]. Popular machine-learning-based approaches for occupancy forecasting are deep neural network architectures [53], Logistic Regression (LogR) [54, 19], Markov Chains [55], Support Vector Machines (SVM), KNN classifiers, RF classifiers as well as boosted classifiers such as AdaBoost, Light Gradient Boosted Machine (LGBM) and XGBoost [10, 52]. In [56], the authors compared the LSTM and GRU for predicting EV arrival and departure times at EV charging stations, concluding that both models are suitable for the tasks and exhibit similar performance. In [10], the authors evaluated the accuracies of XGBoost, KNN, RF and LogR in charging station occupancy prediction using datasets from two sites and attributed the best overall performance to XGBoost. Predictive accuracies of LGBM, AdaBoost, LogR, RF, SVM, naïve Bayes and KNN in charging station occupancy prediction on a national scale were compared in [52], concluding a favorable performance of the boosted classifiers. The authors in [53] examined a variety of deep neural network architectures based on DFN, LSTM, GRU and CNN as well as LogR, RF, SVM and AdaBoost for charging station occupancy prediction at different time scales and attributed the highest predictive accuracy to a model based on a parallel DFN-LSTM architecture.

In summary, there is only a limited number of studies that investigate deep learning models for EV charging station occupancy prediction. Consequently, our model selection primarily relies on the promising results obtained in EV energy demand forecasting, where a greater number of publications exist. The literature review identifies several deep learning models with promising outcomes in both non-EV-related and EV-related energy demand prediction tasks. The number of identifiable well-performing deep learning models for EV charging station occupancy prediction tasks is comparatively lower. In the upcoming theory chapter, the chosen models will be presented, accompanied by two benchmark machine learning algorithms selected for their ease of implementation, low computational costs, and robust performance. Additional references to relevant studies can be found in the sections that discuss each model. Furthermore, the theory chapter will introduce the two selected baseline models.

Chapter 3

Theory

Before discussing how specific machine learning algorithms can be used for time series forecasting, it is important to establish a theoretical framework, including fundamental concepts of supervised learning and time series.

3.1 Supervised Learning

Supervised learning is a machine learning paradigm. A supervised machine learning algorithm is given training data $\mathcal{D}$ as input. This data consists of a collection of pairs $\{(x_i, y_i)\}_{i=1}^{N}$ from $\mathcal{X} \times \mathcal{Y}$, where y_i is called the label corresponding to x_i. The output of the algorithm is a function $\hat{f} : \mathcal{X} \to \mathcal{A}$. For arbitrary $x \in \mathcal{X}$, $\hat{f}$ attempts to predict $y \in \mathcal{Y}$ by choosing an action a from the set of allowed actions, called the *action space* $\mathcal{A}$. The goal of the algorithm is to find a good $\hat{f}$. The quality of a *hypothesis* f proposed by the algorithm is measured by a loss function

$$\mathcal{L} : \mathcal{A} \times \mathcal{Y} \to \mathbb{R}_+,$$

which quantifies the loss incurred from choosing an action a when the true outcome is y. It is assumed that the training data is an independent and identically distributed sample of size N from some unknown joint distribution $\mathbb{P}_{X,Y}$, i.e. the pairs (x_i, y_i) are treated as values of random variables (X_i, Y_i). Further, it is assumed that unseen data points are sampled from the same joint distribution.

If $|\mathcal{Y}|$ is finite, meaning that $y \in \mathcal{Y}$ can only take on a finite number of values, the learning problem is referred to as a *classification problem*. Otherwise, it is called a *regression problem* [57].

The action space $\mathcal{A}$ is introduced because predictions are not necessarily members

of the set $\mathcal{Y}$. For example, to solve a binary classification problem with $\mathcal{Y} = \{0, 1\}$, we could use probabilistic predictions where $\mathcal{A} = [0, 1]$. A prediction is not a direct estimate of y but can be viewed as an action a taken and then judged in the context of y [58].

If we let $\mathcal{A}^{\mathcal{X}}$ denote the set of all functions mapping from $\mathcal{X}$ to $\mathcal{A}$, the problem of finding $\hat{f}$ can be viewed as selecting $\hat{f}$ from $\mathcal{A}^{\mathcal{X}}$ on an empirical basis, which is provided by $\mathcal{D}$.

The true optimal model f^* can be determined by

$$f^* = \underset{f \in \mathcal{A}^{\mathcal{X}}}{\operatorname{argmin}} R(f),$$

where $R(f)$ is called *risk* of a model f and f^* is called the *risk minimizer* [59]. The risk is defined as the expected loss over $\mathbb{P}_{X,Y}$:

$$R(f) = \mathbb{E}[\mathcal{L}(f(X), Y)].$$

f^* can be calculated pointwise as

$$f^*(x) = \underset{f(x) \in \mathcal{A}}{\operatorname{argmin}} \mathbb{E}[\mathcal{L}(f(x), Y) | X = x] \quad \forall x \in \mathcal{X}.$$

which is only possible with complete knowledge of the distribution $\mathbb{P}_{X,Y}$ from which we assume $\mathcal{D}$ was sampled. As we do not know $\mathbb{P}_{X,Y}$, we instead resort to estimating $R(f)$ by calculating the empirical risk $\hat{R}(f)$, defined as

$$\hat{R}(f) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i), y_i). \tag{3.1}$$

We can now find $\hat{f}$, the empirical approximation of f^*, by performing *empirical risk minimization* (ERM):

$$\hat{f} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \hat{R}(f),$$

where $\mathcal{F} \subset \mathcal{A}^{\mathcal{X}}$ is some function space restricted to a subset of the full function space. This restriction is necessary because in the general case, the cardinality of $\mathcal{X}$ is infinite and letting $\mathcal{F} = \mathcal{A}^{\mathcal{X}}$ would require estimating an infinite number of parameters using only a finite data set, which constitutes an ill-posed problem [60].

In machine learning, $\mathcal{F}$ is called *hypothesis space*. Restricting the function space essentially means defining a suitable class of models for solving a given function estimation problem. A popular hypothesis space is the class of linear models

$$\mathcal{F} = \left\{ f : f(x) = \theta_0 + \sum_{i=1}^{p} \theta_i x_i, \quad \forall x \in \mathcal{X} \right\}.$$

The central challenge in machine learning is estimating an $\hat{f}$ with a low true risk $R(\hat{f})$, i.e. a model that performs well on previously unseen data independently sampled from $\mathbb{P}_{X,Y}$. This ability of $\hat{f}$ is referred to as its *generalization* ability. To assess a model's generalization ability, the given data is typically split into a *training set* and a *test set*. Once a model $\hat{f}$ is fit by ERM on the training set, its *generalization error* is given by its empirical risk on the test set. How well a given machine learning algorithm performs is determined by its ability to fit models which make the *training error* $\hat{R}(\hat{f})$ small and the difference between training and generalization error small. Failure to achieve the former objective is denoted as *underfitting*, while not achieving the latter objective is called *overfitting*. Any modification of a machine learning algorithm that aims to reduce its generalization error but not its training error is referred to as *regularization* [61]. Concrete examples of regularization procedures are discussed in Section 3.3.6.

3.2 Fundamentals of Time Series

A *time series* is a sequence of observations ordered in discrete time. A given time series $\mathbf{x} = \{x_t\}_{t=1}^{T} = (x_1, x_2, ..., x_T)$ is a realization of a *stochastic process* $\mathbf{X} = \{X_t\}_{t=1}^{\infty}$ and represents the temporal evolution of an underlying dynamical system. A time series is called *discrete-valued* if it is a realization of a discrete-valued stochastic process and *numeric* if $x_t \in \mathbb{R}^m$, where $m \geq 1$ for $t = 1, ..., T$. If a time series consists of more than one observation per time step, it is called *multivariate*, and *univariate* otherwise [62].

3.2.1 Time Series Forecasting

Given a univariate numeric time series $\mathbf{x} = \{x_t\}_{t=1}^{T}$, *forecasting* is the task of predicting a sequence of h future values. A *window* of p past values is used to make the prediction, h is referred to as the forecasting *horizon*. The forecasting error increases with the horizon h, making multistep forecasting a harder task than step-wise forecasting [63].

We explore two approaches to how supervised learning algorithms can be used for time series forecasting. While Section 3.4 introduces *sequence modeling*, for now, we

focus on how time series forecasting can be formulated as a generic supervised machine learning problem. To this means, a dataset $\mathcal{D}$ has to be constructed from a given time series $\{x_t\}_{t=1}^{T}$. This can be accomplished via time delay embedding [64], which embeds the time series in a Euclidean space of dimension p. The result is a sequence of pairs $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$ from $\mathbb{R}^p \times \mathbb{R}^{h-1}$, where

$$\begin{aligned}\mathbf{x}_i &= (x_{i-p}, x_{i-p+1}, ..., x_{i-1}, x_i)^\top,\\ \mathbf{y}_i &= (x_{i+1}, x_{i+2}, ..., x_{i+h-1})^\top,\\ N &= T - p + h - 1.\end{aligned}$$

If m additional input features have to be included, given as a multivariate time series $(\mathbf{f}_1, \mathbf{f}_2, ..., \mathbf{f}_T)$ with $\mathbf{f}_i \in \mathbb{R}^m$ for $i = 1, ..., T$, the above method is extended by simply concatenating each embedded training example with its corresponding feature vector $\mathbf{f}_i$, resulting in a sequence of pairs $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$ from $\mathbb{R}^{p+m} \times \mathbb{R}^{h-1}$, where

$$\begin{aligned}\mathbf{x}_i &= (x_{i-p}, x_{i-p+1}, ..., x_{i-1}, x_i, f_{i,1}, f_{i,2}, ..., f_{i,m})^\top,\\ \mathbf{y}_i &= (x_{i+1}, x_{i+2}, ..., x_{i+h-1})^\top,\\ N &= T - p + h - 1.\end{aligned}$$

The data $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$ can now be used as input for a supervised regression task as introduced in Section 3.1.

3.2.2 Time Series Classification

Classification is the task of predicting a class label for a given time series. One-step forecasting of discrete-valued binary time series can be viewed as a time series classification problem [65]. In order to formulate it as a supervised machine learning problem, we need to construct a dataset $\mathcal{D}$. Given a discrete binary time series $\{x_t\}_{t=1}^{T}$ and a window size p, the time delay embedding procedure results in a sequence of pairs $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ from $\{0,1\}^p \times \{0,1\}$, where

$$\begin{aligned}\mathbf{x}_i &= (x_{i-p}, x_{i-p+1}, ..., x_{i-1}, x_i)^\top,\\ y_i &= x_{i+1},\\ N &= T - p.\end{aligned}$$

The feature space can be extended analogously to section 3.2.1. The obtained data $\mathcal{D}$ can now be used as input for a supervised classification task as introduced in Section 3.1.

3.2.3 Time Series Analysis

The discipline that encompasses both time series forecasting and classification is time series *analysis*. It aims at extracting useful statistical properties and insights from the study of past observations which can correctly characterize and identify the underlying structure of a given series [66]. In order to perform inference tasks such as forecasting and classification, it is important to know whether the underlying distribution of the observed data changes over time. A time series is said to be *stationary* if the statistical properties of its underlying stochastic process do not change over time. Given a stochastic process $\mathbf{X} = \{X_t\}_{t=1}^{\infty}$, the properties commonly investigated to determine stationarity of a given time series are its moments:

- Mean: $\mu_t = \mathbb{E}[X_t]$
- Variance: $\sigma_t^2 = Var(X_t)$
- Autocovariance: $\gamma_{(X_{t+\tau}, X_t)} = cov(X_{t+\tau}, X_t)$.

Definition 3.2.1 (Strong Stationarity [22])**.** A stochastic process $\mathbf{X}$ is called *strongly stationary* if its joint probability distribution satisfies

$$\mathbb{P}(X_1, X_2, ..., X_T) = \mathbb{P}(X_{1+\tau}, X_{2+\tau}, ..., X_{T+\tau}) \quad \forall T, \tau \in \mathbb{N},$$

i.e. the distribution is invariant to any time shift τ, commonly referred to as *lag*. Every i.i.d. stochastic process is strongly stationary.

Because it is hard to test for strong stationarity, a weak variant called *weak stationarity* is commonly used in practice. It merely constrains the first moment and the autocovariance function of $\mathbf{X}$ instead of its entire distribution.

Definition 3.2.2 (Weak Stationarity [67])**.** A stochastic process $\mathbf{X}$ is called weakly stationary if

1. $\mathbb{E}[X_t] = \mu_{X_t} = \mu < \infty \quad \forall t \in \mathbb{N}$ (mean stationarity)
2. $\gamma_{(X_{t+\tau}, X_t)} = \gamma_\tau \implies Var(X_t) = \gamma_0 < \infty \quad \forall t \in \mathbb{N}.$ (covariance stationarity)

In the following, weak stationarity will be referred to as stationarity for simplicity reasons. Figure 3.1 illustrates stationarity and non-stationarity on simulated time series.

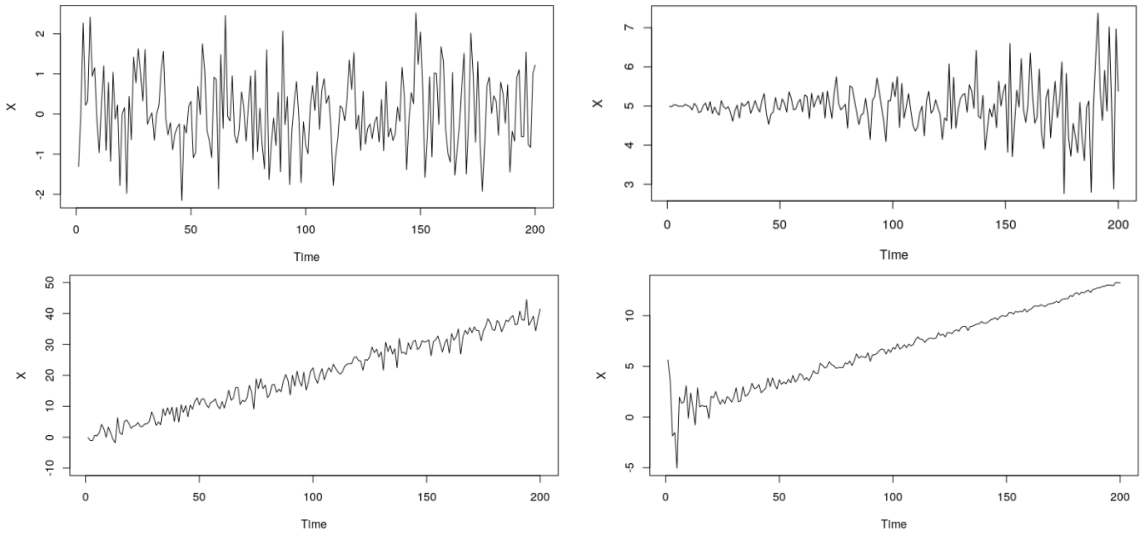


Figure 3.1: Plots of simulated time series: weak stationarity (upper left), mean stationarity and variance non-stationarity (upper right), mean non-stationarity and variance stationarity (lower left), mean and variance non-stationarities (lower right) [26].

A number of widely used statistical forecasting methods are only suitable for stationary time series. These include Autoregressive Integrated Moving Average (ARIMA), Exponential Smoothing (ES), Seasonal Autoregressive Integrated Moving Average (SARIMA) and Vector Autoregression (VAR) models [68].

A common assumption made when analyzing a given non-stationary time series $\{y_t\}_{t=1}^{T}$, for example with the goal of predicting its future values, is that it can be decomposed into a stochastic constituent $\{x_t\}_{t=1}^{T}$, which is stationary, and further components [69]:

- Trend component: $m_t = m(t)$, a usually slowly changing function of time
- Seasonal component: $s_t = s(t)$, a periodic function of time, representing short-term cycles with known frequency
- Cyclic component: $c_t = c(t)$, a periodic function of time, representing long term-term cycles with unknown frequency.

Depending on the observations made, modeling an observed time series of length T can either follow an additive approach

$$y_t = m_t + s_t + c_t + y_t, \quad t = 1, ..., T,$$

a multiplicative approach

$$y_t = m_t s_t c_t x_t, \quad t = 1, ..., T,$$

or a mixed approach

$$y_t = m_t s_t c_t + x_t, \quad t = 1, ..., T.$$

Real-world time series are often non-stationary and display trend, seasonal and noise attributes simultaneously [48]. A straightforward method of assessing the stationarity of a given time series is by visually analyzing its plot over time, which can reveal seasonality, trend and cyclic components as well as changes in variance or unusual features, such as a sudden single peak or a turning point [68].

A method to identify seasonal components of a given time series $\{x_t\}_{t=1}^{T}$ is inspecting its frequency spectrum [70]. To this means, we first compute the discrete Fourier transform (DFT), given by

$$X_k = \sum_{t=1}^{T} x_t e^{-2\pi i(k-1)(t-1)/T}, \quad k = 1, ..., T,$$

where i is the imaginary unit. The magnitude of X_k gives the amplitude of the frequency component at index k. We can then plot the magnitudes of X_k against the frequency index k to visualize the frequency spectrum of the time series.

Oftentimes, it is necessary to use more rigorous statistical tests to determine stationarity. A discussion of time series stationarity tests is beyond the scope of this thesis and can be found in [71].

When using machine learning models to forecast non-stationary time series, which essentially violate the i.i.d.-assumption from Section 3.1, there are two main strategies. The first consists of using architectures specifically designed to learn patterns from sequential data, including long-term dependencies, while the second strategy assumes that a sufficiently complex model can learn non-stationary patterns from incorporated features in the training data that account for the non-stationarity of a given time series. These approaches can also be combined, and both are discussed in this work.

3.2.4 Forecasting Strategies

Predicting multiple time steps ahead is still an open challenge in time series forecasting. The problem is generally approached by using the following strategies:

1. Recursive strategy
2. Direct strategy
3. Multi-input multi-output strategy (MIMO)
4. Combinations the aforementioned strategies.

The recursive forecasting approach is the oldest and most intuitive multi-step prediction method currently in use. In this strategy, a single model f is trained on a given time series $\{x_t\}_{t=1}^{T}$ to perform a one-step forecast, i.e.

$$x_{t+1} = f(x_{t-p}, ..., x_t),$$

where p denotes the embedding dimension as discussed in Section 3.2.1. When forecasting h steps ahead, the first step is obtained by applying the model and the forecasted value is subsequently used as part of the input variables for predicting the next step, using the same model f. Recursive forecasting is sensitive to the accumulation of errors, especially when the horizon h exceeds the embedding dimension p, meaning that at some point, all the inputs to the model are forecasted values instead of real observations. Despite this limitation, the recursive strategy has been successfully used for forecasting tasks on many real-world time series using different machine learning approaches [72].

When multi-step forecasting is carried out using the direct strategy [73], h dedicated models are learned from the time series $\{x_t\}_{t=1}^{T}$ and the forecasts are given by

$$x_{t+i} = f_i(x_{t-p}, ..., x_t), \quad i = 1, ..., h.$$

This strategy does not rely on approximated values, which makes it immune to the accumulation of errors. However, as the h models are learned independently, the direct strategy induces a conditional independence on the h forecasts, which prevents the strategy from considering complex dependencies between the given observations. A further disadvantage of the direct approach is its high computational cost.

While the previous strategies constitute single-output approaches as they model the data as a single-output function, the MIMO strategy learns one multiple-output model f from a given time series $\{x_t\}_{t=1}^{T}$, where the forecast is given by

$$(x_{t+1}, ..., x_{t+h}) = f(x_{t-p}, ..., x_t).$$

The MIMO approach aims to preserve the stochastic dependencies which characterize a given time series, thus avoiding the conditional independence assumption made in the direct approach. Additionally, it avoids the error accumulation problem of recursive forecasting. However, models relying on the MIMO strategy can be susceptible to overfitting and difficult to interpret. So far, MIMO has been successfully applied to several multi-step ahead forecasting tasks on real-world time series [74].

A number of further approaches for multistep-ahead time series forecasting exist, such as the combination of the recursive and the direct strategy (DirRec) as well as DIRMO, which combines the MIMO with the direct approach. A detailed discussion of these strategies can be found in [72].

3.3 Fundamentals of Neural Networks

Deep learning is a subfield of machine learning built around the theory that a deep, hierarchical model can be exponentially more efficient at representing functions than a shallow one [75]. This theory is supported by findings in [76] and [77]. *Feedforward neural networks* (FFNs) and their deep variants, known as *deep feedforward neural networks* (DFNs), represent the prototypical model architectures used in deep learning.

When used for supervised learning, a neural network is given a training dataset $\mathcal{D}$ from $\mathbb{R}^n \times \mathbb{R}^m$ and estimates the optimal model f^* by choosing a hypothesis f from a hypothesis space $\mathcal{F}$ and finding the corresponding empirical risk minimizer $\hat{f}$. The hypothesis is a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ and is composed of a set of parametrized subfunctions $\mathbb{L} = \{f_1, f_2, ..., f_L\}$ that are applied consecutively to the input $\mathbf{x}$. The order in which they are applied is given by a directed acyclic graph $\mathcal{G}$ called *network topology*. The nodes of $\mathcal{G}$ represent the DFN's layers and its directed edges indicate whether the output of one layer is the input of another layer. If a node has only outgoing edges, it represents the input $\mathbf{x} \in \mathbb{R}^n$ while nodes without outgoing edges are the network's output $f(\mathbf{x}) \in \mathbb{R}^m$. $\mathcal{G}$ is acyclic, meaning no loop in the graph can be traversed without violating the orientation of at least one edge. This property is eponymous for feedfor-

ward neural networks, as information is only propagated in one direction through $\mathcal{G}$ [61]. Neural networks with an underlying cyclic topology will be discussed in Section 3.4.

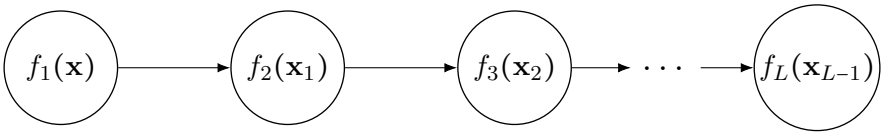


Figure 3.2: Simple DFN with layers $f_1, ..., f_L$, where $f_1(\mathbf{x}) = \mathbf{x} =: \mathbf{x}_1$ (input layer) and $\mathbf{x}_{l+1} := f_{l+1}(\mathbf{x}_l)$.

DFNs are universal function approximators and can theoretically represent any function [78]. They are a class of models that use parameters $\boldsymbol{\theta}$ in order to estimate $\hat{f}$:

$$\hat{f}(\mathbf{x}) = f(\mathbf{x}; \hat{\boldsymbol{\theta}}).$$

Commonly, the optimal parameters $\hat{\boldsymbol{\theta}}$ are estimated or *learned* from $\mathcal{D}$ by minimizing the empirical risk

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \hat{R}(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i), \tag{3.2}$$

where $\mathcal{L} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}_+$ is a differentiable loss function. The process of estimating optimal model parameters is referred to as *training* the neural network. Training consists of three fundamental steps, which are explored in the following sections:

1. Forward pass
2. Backpropagation
3. Parameter update via gradient descent.

3.3.1 Forward Pass

Forward pass and backpropagation in DFNs are explained by means of the prototypical layer used in neural networks, the *fully connected* or *dense* layer.

A dense layer $f_l \in \mathbb{L}$ can be viewed as a complete directed bipartite graph with nodes contained in two disjoint sets of sizes n_{l-1} and n_l. These nodes represent the inputs and the outputs of the layer respectively, while the directed edges connect every input node to all output nodes of f_l. The edges are referred to as the weights of the layer [79].

A mapping

$$\begin{aligned} f_l : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}, \quad \mathbf{x} \mapsto (a_1(\mathbf{x}), ..., a_{n_l}(\mathbf{x})), \\ \text{where } a_j := \phi_l(\mathbf{w}_j^\top \mathbf{x} + b_j), \end{aligned} \tag{3.3}$$

with a weight vector $\mathbf{w}_j \in \mathbb{R}^{n_{l-1}}$ and a bias term $b_j \in \mathbb{R}$ for $j = 1, ..., n_l$ and some nonlinear function $\phi_l : \mathbb{R} \to \mathbb{R}$ is called fully connected or dense layer. $\mathbf{w}_j^\top \mathbf{x}$ is called weighted input and a_j the activation of the j-th node of the layer. ϕ_l is referred to as the activation function of layer l.

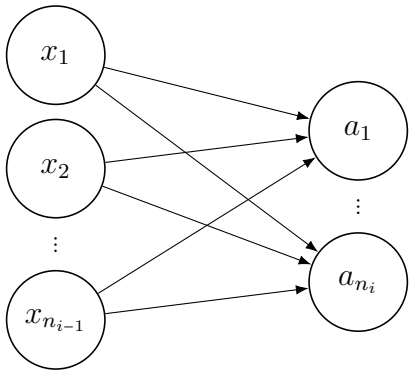


Figure 3.3: Fully connected layer f_i.

An overview of the nonlinear activation functions used in neural networks within this work is given in Table 3.1. Intuitively speaking, the nodes mimic the behavior of biological neurons by collecting all node activations from the previous layer scaled by their respective weights and calculating an activation, which gets close to one (i.e. "fires") only if $\mathbf{w}_j^\top \mathbf{x}$ is large. The bias term is added as a threshold mechanism: The neuron only "fires" if $\mathbf{w}_j^\top \mathbf{x} > -b_j$.

f_1 is the network's *input layer* and f_L is the *output layer*. The layers $f_2, ..., f_{L-1}$ are called *hidden layers*. A deep FFN is defined as having multiple nonlinear hidden layers between input and output.

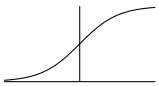
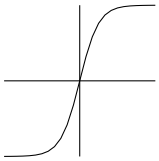
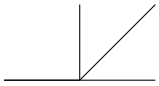
Name	Function	Derivative	Plot
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))^2$	
Tanh	$tanh(x) = \frac{e^x - e^{-x}}{e^z + e^{-z}}$	$f'(x) = 1 - f(x)^2$	
ReLU	$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0. \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 1 \geq 0. \end{cases}$	

Table 3.1: Nonlinear activation functions used in neural networks within this thesis.

Neural networks are a kind of *adaptive basis function models*, which use *basis function expansion* to extend the class of linear models introduced in Section 3.1 to the more general nonlinear case [80]. Basis function expansion models take the form

$$f(x) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x), \tag{3.4}$$

where ϕ_i are potentially nonlinear functions of x for $i = 1, ..., M$. As the name suggests, adaptive basis function models constitute a special case of Equation 3.4 in that the basis functions do not take a predetermined form and are learned from data. Both the entire network and every single layer can be viewed as an adaptive basis function model. If we let $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ denote the matrix of weights and $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ denote the vector of biases for layer l, the dense layer f_l can be expressed as

$$f_l(\mathbf{x}) = \phi_l(\mathbf{W}^{(l)}\mathbf{x} + \mathbf{b}^{(l)}),$$

where ϕ_l is applied element-wise. By letting a_j^l denote the output or activation from neuron j at layer l and $z_j^{(l)}$ the weighted input for neuron j at layer l, we can summarize the actions performed by a fully connected DFN $f : \mathbb{R}^n \to \mathbb{R}^m$ during forward pass:

$$\mathbf{a}^1 = \mathbf{x} \in \mathbb{R}^{n_1}, \quad \text{where } n_1 = n,$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \in \mathbb{R}^{n_l} \quad \text{for } l = 2, 3, ..., L,$$

$$\mathbf{a}^{(l)} = \phi_l(\hat{\mathbf{z}}^{(l)}) \in \mathbb{R}^{n_l} \quad \text{for } l = 2, 3, ..., L.$$

An expression analogous to Equation 3.4 for the entire network is thus given by

$$f(\mathbf{x}) = \mathbf{b}^{(L)} + \sum_{j=1}^{n_L} \mathbf{w}_j^{(L)} \phi_j(\mathbf{x})$$

$$= \mathbf{b}^{(L)} + \sum_{j=1}^{n_L} \mathbf{w}_j^{(L)} \phi_j \left(\mathbf{b}^{(L-1)} + \sum_{i=1}^{n_{L-1}} \mathbf{w}_i^{(L-1)} \phi_i \left(\mathbf{b}^{(L-2)} + \sum_{k=1}^{n_{L-2}} \mathbf{w}_k^{(L-2)} \phi_k \left(\cdots \right) \right) \right),$$

and we can see that a DFN is a composition of adaptive basis function models.

Two further important and widely used supervised learning algorithms based on adaptive basis function models that are discussed within this work are tree models and boosted models.

The following sections focus on how DFNs learn by adjusting the weight vectors and the bias terms.

3.3.2 Backpropagation

Finding parameters $\hat{\boldsymbol{\theta}}$ that minimize a neural network's empirical risk is commonly accomplished by *gradient descent*, a first-order iterative optimization algorithm which is discussed in the next section. It makes use of the gradient of the objective function with respect to the network's parameters. In DFNs, this gradient is typically computed by the *backpropagation* algorithm, a dynamic programming procedure that exploits the properties of the chain rule of calculus. Backpropagation in neural networks is a special case of reverse-mode *automatic differentiation*, which provides a general framework to compute the derivatives of any function $f : \mathbb{R}^n \to \mathbb{R}^m$ that can be evaluated with a differentiable computational graph[1] [81]. Backpropagation is a special case, as it computes derivatives of the scalar-valued loss function with respect to the network's parameters.

[1] A computational graph is a directed acyclic graph $G = (\{1, ..., N\}, E)$ with associated variables $u_1, ..., u_N$ of dimensionalities $d_1, ..d_N$ and with local functions $f_1, ...f_n$. Vertices with no outgoing edges are called output vertices and vertices without incoming edges are called input vertices. The set of input variables of vertex i is defined as $\alpha_i = \{u_i | (j, i) \in E\}$. If i is an input vertex, this set is empty [57].

To illustrate how backpropagation makes use of the chain rule, we consider some differentiable function

$$J(x) = f(g(h(x))),$$

$$\text{with } z_1 = h(x),\ z_2 = g(z_1),\ z_3 = f(z_2).$$

To compute its partial derivatives, we could naïvely derive the respective expressions and compute all partial derivatives via

$$\frac{\partial J(x)}{\partial x} = \frac{\partial z_1}{\partial x}\frac{\partial z_2}{\partial z_1}\frac{\partial z_3}{\partial z_2}, \tag{3.5}$$

$$\frac{\partial J(x)}{\partial z_1} = \frac{\partial z_2}{\partial z_1}\frac{\partial z_3}{\partial z_2}, \tag{3.6}$$

$$\frac{\partial J(x)}{\partial z_2} = \frac{\partial z_3}{\partial z_2}, \tag{3.7}$$

which would pose a high computational cost if the number of function compositions increased. To be more efficient, we can leverage the repeating pattern in the above expressions which arises from the relations between the partial derivatives of x, z_1 and z_2. By using the definition of $\frac{\partial J(x)}{\partial z_1}$, we can rewrite Equation 3.5 as

$$\frac{\partial J(x)}{\partial x} = \frac{\partial z_1}{\partial x}\frac{\partial J(x)}{\partial z_1},$$

and get only one new partial derivative per step into the chain of functions. The remaining partial derivates are already computed if we compute them in the order z_2, z_1, x, requiring only one derivative computation per parameter, while the number of derivatives to be calculated grows linearly in the naïve approach [82].

This efficient procedure can be directly transferred to the computation of gradients in DFNs: Because partial derivatives w.r.t to a single weight in layer l depend on partial derivatives w.r.t weights in layer $l+1$, by using the appropriate overall computation order, a gradient computation carried out in layer l can reuse the results of gradient computations from layer $l+1$. This computation order is eponymous for the algorithm: Gradients are computed starting at the output layer and then propagated *backward* through the network.

The backpropagation algorithm computes the gradient of a neural network's objective function with respect to the network's parameters. Starting at layer L, it computes the gradient for a fixed pair $(\mathbf{x}_i, \mathbf{y}_i)$ with respect to the output of the layer and then, via the chain rule, with respect to the layer's parameters. For a differentiable objective function $\hat{R}(\boldsymbol{\theta})$, we have

$$\nabla \hat{R}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i),$$

where we used Equation 3.2.

In order to derive expressions for the gradients of the loss function with respect to the network's learnable parameters $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ for each layer $l = 1, 2, ...L$, we introduce the quantity

$$\delta_j^{(l)} := \frac{\partial \mathcal{L}}{\partial z_j^{(l)}},$$

called the *error signal* in the j-th neuron in layer l. For the output layer L, the components of $\boldsymbol{\delta}^{(L)} \in \mathbb{R}^{n_L}$ are computed by applying the chain rule:

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \phi_L'(z_j^{(L)}). \tag{3.8}$$

The expression for the components of $\boldsymbol{\delta}^{(l)} \in \mathbb{R}^{n_l}$ in terms of the error signal in the next layer $\boldsymbol{\delta}^{(l+1)}$ can be derived from

$$\begin{aligned} z_k^{(l+1)} &= \sum_{j}^{m_{l+1}} w_{kj}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} = \sum_{j}^{m_{l+1}} w_{kj}^{(l+1)} \phi_l(z_j^{(l)}) + b_k^{(l+1)} \\ \implies \quad & \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = w_{kj}^{(l+1)} \phi_l'(z_j^{(l)}). \end{aligned} \tag{3.9}$$

By applying the chain rule to the equation above, we get

$$\begin{aligned} \delta_j^{(l)} &= \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \sum_{k=1}^{m_{l+1}} \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_k^{(l)}} \\ &= \sum_{k=1}^{m_{l+1}} \frac{\partial z_k^{(l+1)}}{\partial z_k^{(l)}} \delta_k^{(l+1)} = \sum_{k=1}^{m_{l+1}} w_{kj}^{(l+1)} \delta_k^{(l+1)} \phi_l'(z_j^{(l)}). \end{aligned} \tag{3.10}$$

To compute $\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}}$, we keep in mind that

$$z_j^{(l)} = \sum_k^{m_l} w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)} \quad \Longrightarrow \quad \frac{\partial z_j^{(l)}}{\partial c_{jk}^{(l)}} = a_k^{(l-1)}. \tag{3.11}$$

We now apply the chain rule and get

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)}, \tag{3.12}$$

As for the partial derivative $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}}$, we consider

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial b_j^{(l)}} \frac{\partial b_j^{(l)}}{\partial z_j^{(l)}}, \tag{3.13}$$

and get

$$z_j^{(l)} = \sum_k^{m_l} c_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)} \quad \Longrightarrow \quad \frac{\partial b_j^{(l)}}{\partial z_j^{(l)}} = 1 \quad \Longrightarrow \quad \frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)}. \tag{3.14}$$

Equations 3.8, 3.10, 3.12 and 3.14 can be written in vectorized form as follows [83]:

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)}} \odot \phi_L'(\mathbf{z}^{(L)}), \quad \boldsymbol{\delta}^{(l)} = (\mathbf{W}^{(l+1)})^\top \boldsymbol{\delta}^{(l+1)}) \odot \phi_l'(\mathbf{z}^{(l)}), \tag{3.15}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\mathbf{a}^{(l-1)})^\top, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} \boldsymbol{\delta}^{(l)}, \tag{3.16}$$

where the activation functions ϕ_i are applied elementwise and "$\odot$" denotes the Hadamard product. Moreover, we made use of matrix calculus notation, where the derivative of a scalar f with respect to a vector $\mathbf{x} \in \mathbb{R}^n$ is written as

$$\frac{\partial f}{\partial \mathbf{x}} = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n} \right)^\top,$$

and the derivative of a scalar with respect to a matrix is defined by a corresponding matrix of elementwise derivatives [83].

If we use squared error loss (SEL), the most common loss function for regression tasks, given by

$$\mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{2}\|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2, \quad \text{where} \quad \hat{\mathbf{y}}_i = f(\mathbf{x}_i),$$

we get

$$\boldsymbol{\delta}^{(L)} = (\hat{\mathbf{y}}_i - \mathbf{y}_i) \odot \phi_L'(\mathbf{z}^{(L)}).$$

Equations 3.15 allow us to compute the error $\boldsymbol{\delta}^{(l)}$ for any layer in the network and constitute the backpropagation algorithm. As the calculation of gradients only requires matrix multiplications, the algorithm can be implemented efficiently in most programming languages.

The calculations performed during forward pass and consecutive error backpropagation in a DFN for one training example $(\mathbf{x}_i, \mathbf{y}_i)$ are outlined in Algorithm 1.

Algorithm 1: Forward Pass & Backpropagation

Data: Training example $(\mathbf{x}_i, \mathbf{y}_i)$

Result: Gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\mathbf{a}^{(l-1)})^\top$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$

Set input layer activation $\mathbf{a}^1 = \mathbf{x}_i$

for $l = 2, ..., L$ **do**

  $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$

end

$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)}} \odot \phi_L'(\mathbf{z}^{(L)})$

for $l = L - 1, ..., 2$ **do**

  $\boldsymbol{\delta}^{(l)} = (\mathbf{W}^{(l+1)})^\top \boldsymbol{\delta}^{(l+1)}) \odot \phi_l'(\mathbf{z}^{(l)})$

end

In the final step of learning, the network parameters are updated using gradient descent, which is discussed in the next section.

3.3.3 Gradient Descent

Gradient descent algorithms are a class of first-order iterative optimization algorithms and by far the most commonly used method for optimizing neural networks [84]. For neural network models, numerical optimization is performed in parameter space. Optimization procedures in function space will be discussed in Section 3.13. Gradient descent updates the network parameters in the direction of steepest empirical risk descent, which is the opposite direction of the gradient of the objective calculated during backpropagation. In other words, learning is performed by following the direction of the slope of the surface created by the objective function until a critical point of $\hat{R}$ is reached where the gradient vanishes, e.g. a (local) minimum.

At iteration n of gradient descent, the current estimate of the parameters, denoted by $\boldsymbol{\theta}^{(n-1)}$, is updated by taking a step $\boldsymbol{\theta}_n$ in the direction of steepest descent of the risk $\hat{R}$, given by the scaled negative gradient $-\gamma_n \mathbf{g}_n$, i.e.

$$\mathbf{g}_n = \nabla \hat{R}(\boldsymbol{\theta}^{(n-1)}), \tag{3.17}$$

$$\boldsymbol{\theta}_n = -\gamma_n \mathbf{g}_n, \tag{3.18}$$

$$\boldsymbol{\theta}^{(n)} = \boldsymbol{\theta}^{(n-1)} + \boldsymbol{\theta}_n. \tag{3.19}$$

where $\gamma_n \in \mathbb{R}_+$ denotes the step size or learning rate. If we let $\boldsymbol{\theta}_0$ denote some initial guess, the final parameter estimate after M iterations is given by

$$\boldsymbol{\theta}^{(M)} = \sum_{n=0}^{M} \boldsymbol{\theta}_n.$$

For a *convex*[2] and *L-smooth*[3], empirical risk function $\hat{R} : \mathbb{R}^D \to \mathbb{R}$, where D denotes the total number of parameters of a neural network, gradient descent with a fixed step size of $\gamma_n \leq \frac{1}{L}$ is guaranteed to converge to $\hat{\boldsymbol{\theta}}$, and the parameter estimate $\boldsymbol{\theta}^{(n)}$ satisfies

$$\hat{R}\left(\boldsymbol{\theta}^{(n)}\right) - \hat{R}(\hat{\boldsymbol{\theta}}) = O\left(1/n\right),$$

where n denotes the number of steps taken. A proof is given in [85].

[2] A function $f : \mathbb{R}^d \to \mathbb{R} \cup \{+\infty\}$ is *convex* if for all $x, y \in \mathbb{R}^d$, for all $t \in [0,1]$, we have $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$.

[3] A function $f : \mathbb{R}^d \to \mathbb{R}$ is *L-smooth* if it is differentiable and if $\nabla f : \mathbb{R}^d \to \mathbb{R}^d$ is *L-Lipschitz*, i.e for all $x, y \in \mathbb{R}^d$, we have $\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$ [85].

The convergence rate implies that in order to achieve a bound of

$$\hat{R}(\boldsymbol{\theta}^{(n)}) - \hat{R}(\hat{\boldsymbol{\theta}}) \leq \epsilon,$$

gradient descent has to run for $O\left(1/\epsilon\right)$ iterations, which is referred to as *sub-linear* convergence. For *strongly convex*[4] $\hat{R}$, the same bound can be achieved using only $O\left(\log 1/\epsilon\right)$ iterations, which is referred to as *linear* convergence [86].

In neural networks, the surfaces of objective functions are typically highly non-convex and the criterion landscape has many local minima as well as saddle points, i.e. points where the gradient vanishes that are not local minima. In high dimensions, it is possible to construct functions where gradient descent almost always gets stuck at a saddle point. Finding global minima of high-dimensional highly-non-convex functions is generally hard. In practice, only a small number of local minima are actually bad solutions for a given supervised learning task, and most optimization techniques used by machine learning practitioners focus on approximating at least one local minimum of the objective function *efficiently* [86].

In order to explore how different flavors of gradient descent compare in their computational efficiency, we define the following function for more convenient notation:

$$\mathcal{L}_i : \mathbb{R}^D \to \mathbb{R}, \quad \boldsymbol{\theta} \mapsto \mathcal{L}\left(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i\right). \tag{3.20}$$

Using the notation introduced in the previous sections and considering that

$$\nabla \hat{R}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{L}_i(\boldsymbol{\theta}),$$

we can formulate one variant of a full training procedure of a DFN, outlined in Algorithm 2, where we apply the update rule from Equation 3.19.

[4] Let $f : \mathbb{R}^d \to \mathbb{R} \cup \{+\infty\}$ and $\alpha > 0$. f is called α-*strongly convex* if for all $x, y \in \mathbb{R}^d$, for all $t \in [0, 1]$, we have $\alpha \frac{t(1-t)}{2} \|x - y\|^2 + f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$ [85].

Algorithm 2: Neural network training via Batch Gradient Descent

Data: Training dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$

for $i = 1, ..., N$ **do**
  Set input activation $\mathbf{a}^{i,1} = \mathbf{x}_i$
  for $l = 2, ..., L$ **do**
    $\mathbf{z}^{i,l} = \mathbf{W}^{(l)}\mathbf{a}^{i,l-1} + \mathbf{b}^{(l)}$
  end
  $\boldsymbol{\delta}^{i,L} = \frac{\partial \mathcal{L}_i}{\partial \mathbf{a}^{i,L}} \odot \phi_L'(\mathbf{z}^{i,L})$
  for $l = L-1, ..., 2$ **do**
    $\boldsymbol{\delta}^{i,l} = (\mathbf{W}^{(l+1)})^\top \boldsymbol{\delta}^{i,l+1}) \odot \phi_l'(\mathbf{z}^{i,l})$
  end
end
for $l = L, L-1, ..., 2$ **do**
  $\mathbf{w}_n^{(l)} = \mathbf{w}_{n-1}^{(l)} - \frac{\gamma_n}{N} \sum_{i=1}^{N} \boldsymbol{\delta}^{i,l} (\mathbf{a}^{i,l-1})^T$
  $\mathbf{b}_n^{(l)} = \mathbf{b}_{n-1}^{(l)} - \frac{\gamma_n}{N} \sum_{i=1}^{N} \boldsymbol{\delta}^{i,l}$
end

In this algorithm, the whole training dataset is used to perform just one parameter update, and it is referred to as *batch gradient descent*. It repeats the following steps:

$$\boldsymbol{\theta}^{(n)} = \boldsymbol{\theta}^{(n-1)} - \frac{\gamma_n}{N} \sum_{i=1}^{N} \nabla \mathcal{L}_i \left(\boldsymbol{\theta}^{(n-1)} \right),$$

and can be very costly for large datasets, as gradients for the entire dataset have to be computed during each iteration [87].

A computationally much cheaper alternative is to replace the mean of the individual gradients over all training examples with the gradient of a single training example. This can be done by following two different approaches [86]:

1. Choose i_n uniformly at random with replacement from $\{1, 2, ..., N\}$ at each iteration n (randomized approach).

2. Choose i_n following the pattern $1, 2, ..., N, 1, ..., N, ...$ (cyclic approach).

This represents the simplest form of a procedure called *stochastic gradient descent* (SGD). Given an appropriately chosen i_n, we can summarize an update step of SGD by

$$\mathbf{g}_n^{(i_n)} = \nabla \mathcal{L}_{i_n} \left(\boldsymbol{\theta}^{(n-1)} \right),$$

$$\boldsymbol{\theta}^{(n)} = \boldsymbol{\theta}^{(n-1)} - \gamma_n \mathbf{g}_n^{(i_n)}.$$

The randomized approach is more common in practice, for which we have

$$\mathbb{E}\left[\nabla \mathcal{L}_{i_n}\left(\boldsymbol{\theta}^{(n-1)}\right)\right] = \sum_{i=1}^{N} \nabla \mathcal{L}_i\left(\boldsymbol{\theta}^{(n-1)}\right),$$

i.e. the stochastic gradient gives an *unbiased estimate* of the full gradient at each step.

It is standard to use decaying step sizes in SGD, for example $\gamma_n = \frac{1}{n}$. The short explanation, for which we assume the cyclic approach for simplicity, is the following: After N updates of SGD with constant step size $\gamma_n = \gamma$, we have

$$\boldsymbol{\theta}^{(n+N)} = \boldsymbol{\theta}^{(n)} - \gamma \sum_{i=1}^{N} \nabla \mathcal{L}_i\left(\boldsymbol{\theta}^{(n+i-1)}\right),$$

while for a single parameter update in batch gradient descent with step size γN, we have

$$\boldsymbol{\theta}^{(n+1)} = \boldsymbol{\theta}^{(n)} - \gamma \sum_{i=1}^{N} \nabla \mathcal{L}_i\left(\boldsymbol{\theta}^{(n)}\right).$$

and therefore

$$\boldsymbol{\theta}^{(n+N)} - \boldsymbol{\theta}^{(n+1)} = \gamma \sum_{i=1}^{N} \left[\nabla \mathcal{L}_i\left(\boldsymbol{\theta}^{(n+i-1)}\right) - \nabla \mathcal{L}_i\left(\boldsymbol{\theta}^{(n)}\right)\right].$$

For constant γ, this difference will in general not go to zero and hence, a decaying step size will make the stochastic update a more accurate estimate of the full gradient [86]. A more general discussion of decaying learning rates in SGD is given in [85].

For a convex objective function $\hat{R}$, the parameter estimate $\boldsymbol{\theta}^{(n)}$ found by SGD with diminishing step sizes satisfies

$$\mathbb{E}\left[\hat{R}\left(\boldsymbol{\theta}^{(n)}\right)\right] - \hat{R}(\hat{\boldsymbol{\theta}}) = O\left(1/\sqrt{n}\right),$$

and the bound does not improve when further assuming L-smoothness of $\hat{R}$. For strongly convex and L-smooth $\hat{R}$, we have

$$\mathbb{E}\left[\hat{R}\left(\boldsymbol{\theta}^{(n)}\right)\right] - \hat{R}(\hat{\boldsymbol{\theta}}) = O\left(1/n\right),$$

which means that SGD does not exhibit the linear convergence rate of batch gradient descent under strong convexity [88].

In practice, SGD is often able to converge to an acceptable parameter estimate faster than batch gradient descent. Its frequent parameter updates exhibit high variance, causing the objective function to fluctuate heavily. While these fluctuations can enable SGD to jump to new and potentially better local minima, they ultimately complicate convergence as SGD will keep overshooting the exact minimum [87]. However, this is not considered a significant problem due to the fact that full optimization of the objective can result in overfitted models.

A widely used compromise between batch gradient descent and SGD is *mini-batch* SGD. For some randomly chosen set $B_n \subset \{1, 2, ..., N\}$, where $|B_n| = b \ll N$, it takes steps of the following form:

$$\mathbf{g}_n^{(i:i+b)} = \frac{1}{b} \sum_{i \in B_n} \nabla \mathcal{L}_i \left(\boldsymbol{\theta}^{(n-1)} \right), \tag{3.21}$$

$$\boldsymbol{\theta}^{(n)} = \boldsymbol{\theta}^{(n-1)} - \gamma_n \mathbf{g}_n^{(i:i+b)}. \tag{3.22}$$

In this approach, the set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i \in B_n}$ is called a *batch* of size b. As before, the gradient estimate is unbiased, i.e.

$$\mathbb{E}\left[\frac{1}{b} \sum_{i \in B_n} \nabla \mathcal{L}_i \left(\boldsymbol{\theta}^{(n-1)} \right) \right] = \sum_{i=1}^{N} \nabla \mathcal{L}_i \left(\boldsymbol{\theta}^{(n-1)} \right),$$

and its variance reduction by $\frac{1}{b}$ comes at the cost of b times more necessary computations per iteration. As shown in [89], for a convex, L-smooth objective function $\hat{R}$, the parameter estimate $\boldsymbol{\theta}^{(n)}$ found by mini-batch SGD satisfies

$$\mathbb{E}\left[\hat{R}\left(\boldsymbol{\theta}^{(n)} \right) \right] - \hat{R}(\hat{\boldsymbol{\theta}}) = O\left(\sqrt{b/n} + b/n \right).$$

The variance of parameter updates is smaller than in SGD, which can lead to more stable convergence. A further advantage of SGD is that it is trivially parallelizable, which makes it attractive for large-scale machine learning applications. For very large datasets, it is not uncommon for mini-batch SGD to converge in the time that it takes batch GD to perform a single parameter update [81]. When training neural networks, mini-batch SGD is often the optimization algorithm of choice.

Nonetheless, there exist certain scenarios where the algorithm has difficulties. It can get trapped in areas where the gradient is small if the learning rate is also small. Using the same learning rate for all parameter updates can be problematic when handling sparse or heterogeneous data. If the algorithm encounters an area of high curvature it

can take an excessively large step, missing valuable information in the local neighborhood [61].

One recent state-of-the-art optimization algorithm which addresses these challenges is ADAM, which is discussed in the next section.

3.3.4 ADAM

Unstable step directions and oscillations between iterations, as exhibited by mini-batch SGD, can be overcome by retaining information about previous gradients. This is accomplished by the method of *momentum* [90], which incorporates moments of the objective function into its parameter update rule. For a discrete random variable X with finite support, the n-th moment is defined to be

$$\mathbb{E}\left[X^n\right] = \sum_{i=1}^{N} x_i^n \Pr(X = x_i),$$

and the n-th central moment is defined to be

$$\mathbb{E}\left[(X - \mathbb{E}[X])^n\right] = \sum_{i=1}^{N} (x_i - \mathbb{E}[X])^n \Pr(X = x_i).$$

Optimization algorithms based on the method of momentum incorporate the moments of the objective function in their parameter update rule in some way. For example, a method that only uses the first moment can be expressed as

$$\mathbf{g}_n = \nabla \hat{R}\left(\boldsymbol{\theta}^{(n-1)}\right),$$

$$\mathbf{m}_n = \beta \mathbf{m}_{n-1} - \gamma_n \mathbf{g}_n,$$

$$\boldsymbol{\theta}^{(n)} = \boldsymbol{\theta}^{(n-1)} + \mathbf{m}_n,$$

where $\mathbf{m}_n$ is an exponentially decreasing average of past gradients and the momentum decay coefficient $\beta \in [0, 1)$ controls the rate at which old gradients are discarded. γ_n is a scalar learning rate that controls the step size.

ADAM stands for Adaptive Moment Estimation and was introduced in 2014 [91]. It uses adaptive estimates of the first- and second-order moments, denoted by $\mathbf{m}_n$ and $\mathbf{v}_n$, respectively. The update rule is based on exponentially decaying moving averages of the gradient $\mathbf{g}_n$ and the squared gradient $\mathbf{g}_n \odot \mathbf{g}_n$. In the following equations, which show one iteration of ADAM, b denotes the batch size i.e. the number of randomly chosen training examples (analogous to mini-batch SGD), while $\beta_1 \in [0, 1)$ and $\beta_2 \in [0, 1)$

are the decay rates for the moment estimates, and η is called learning rate and has a default range of $[0.001, 0.1]$. $\mathbf{m}_0$ and $\mathbf{v}_0$ are both initialized to $\mathbf{0}$.

$$\mathbf{g}_n = \frac{1}{b} \sum_{i \epsilon B_n} \nabla \mathcal{L}_i \left(\boldsymbol{\theta}^{(n-1)} \right), \tag{3.23}$$

$$\mathbf{m}_n = \beta_1 \mathbf{m}_{n-1} + (1 - \beta_1) \mathbf{g}_n, \tag{3.24}$$

$$\mathbf{v}_n = \beta_2 \mathbf{v}_{n-1} + (1 - \beta_2)\ \mathbf{g}_n \odot \mathbf{g}_n, \tag{3.25}$$

$$\hat{\mathbf{m}}_n = \frac{\mathbf{m}_n}{1 - \beta_1^n}, \tag{3.26}$$

$$\hat{\mathbf{v}}_n = \frac{\mathbf{v}_n}{1 - \beta_2^n}, \tag{3.27}$$

$$\boldsymbol{\theta}^{(n)} = \boldsymbol{\theta}^{(n-1)} - \eta \frac{\hat{\mathbf{m}}_n}{\sqrt{\hat{\mathbf{v}}_n} + \epsilon}. \tag{3.28}$$

In Equations 3.26 and 3.27, a bias correction is applied because $\mathbf{m}_n$ and $\mathbf{v}_n$ are biased towards $\mathbf{0}$ due to their initialization. In Equation 3.28, $\epsilon \approx 10^{-8}$ is added for numerical stability.

According to [87], ADAM performs well on sparse data sets and the authors in [91] show empirically that ADAM compares favorably to other adaptive optimization algorithms. In [92], the authors state that while ADAM exhibits empirically better optimization performance than mini-batch SGD in numerous scenarios, it often suffers from a comparatively poorer generalization performance, especially when used for training DFNs. On the other hand, experiments conducted in [93] showed that when tuned carefully, ADAM and other adaptive gradient methods never underperform mini-batch SGD.

3.3.5 Further Layer Types

A number of models employed for time series forecasting within this thesis make use of *convolutional layers*. In general, neural networks that make use of at least one convolutional layer are referred to as *Convolutional Neural Networks* (CNNs) [61]. They represent a large family of neural networks specially designed to filter and extract features from data with grid-like topology [94]. For example, a univariate time series can be thought of as a 1D grid with a regular sample interval. CNNs are traditionally designed for image datasets and extract local relationships that are invariant to translation [95].

Recently, CNNs utilizing one-dimensional (1D) convolutional layers have been applied successfully for time series forecasting and especially energy demand forecasting tasks [96, 97, 98].

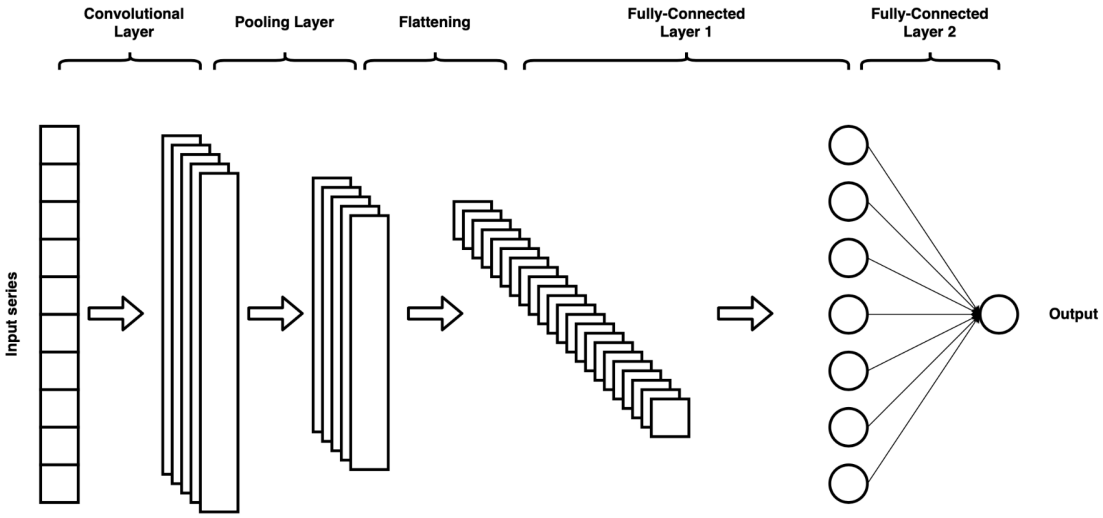


Figure 3.4: Exemplary structure of a 1D convolutional neural network [99].

1D-CNNs typically consist of the following types of layers, see Figure 3.4:

1. 1D convolutional layers
2. Pooling layers
3. Flatten layers
4. Fully connected layers.

The first three layer types will be discussed in the following sections.

3.3.5.1 1D Convolutional Layer

The following discussions are based on [100, 101, 102].

Fully connected neural networks do not scale well when presented with high-dimensional input data such as digital images. CNNs address this problem by constraining the number of values that are allowed in weight matrices and bias vectors. Instead of one full-sized linear transformation, a convolutional layer applies a number of small-scale

linear kernels, or filters, across the input data. The weight matrices used in convolutional layers are sparse and highly structured, implying that in a CNN, each neuron in a layer is only connected to a small local region of neurons in the previous layer, rather than to every neuron. Convolutional layers leverage the idea of parameter sharing: Unlike in fully connected layers, where every element is used exactly once when computing the output, each parameter of a kernel is used at every position of the input, which greatly reduces the number of learnable parameters.

To understand how linear filters work, we can consider multiplying an input vector in $\mathbb{R}^6$ by the matrix

$$\begin{bmatrix} 1 & -1 & & & & \\ & 1 & -1 & & & \\ & & 1 & 1 & & \\ & & & 1 & -1 & \\ & & & & 1 & -1 \end{bmatrix} \in \mathbb{R}^{5\times 6},$$

This operation results in a vector in $\mathbb{R}^5$ made up of differences between neighboring values. This is useful for detecting abrupt changes between neighboring values. The operation uses the filter $[1, -1]$ and a *stride* of one, which means that the filter advances by one entry after each use. By specifying filter size and stride length, we can allow the training process to learn weights in the filter as a means to extract useful features.

A 1D-convolutional layer using one kernel can be expressed just like a fully connected layer:

$$f(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b}),$$

where $\mathbf{b} = (b, ..., b)^\top$ denotes the reduced bias vector and ϕ is a nonlinear activation function. A more common way to express the computations in 1D-convolutional layers makes use of the following concept:

Definition 3.3.1 (Discrete 1D-Cross-Correlation [103])**.** For real-valued functions f and g defined on the set $\mathbb{Z}$ of integers, the discrete cross-correlation (or non-causal convolution) of f and g is given by:

$$(f \star g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(m+n).$$

The cross-correlation of finite sequences is calculated by extending the sequences to finitely supported functions on $\mathbb{Z}$ and using a finite summation.

For example, in the matrix-vector multiplication

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 & & & & \\ & w_1 & w_2 & w_3 & & & \\ & & w_1 & w_2 & w_3 & & \\ & & & w_1 & w_2 & w_3 & \\ & & & & w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ 0 \end{bmatrix},$$

we have $f(-1) = w_1, f(0) = w_2, f(1) = w_3, f(k) = 0$, else and $g(1) = x_1, ..., g(5) = x_5, g(l) = 0$, else. Then $y_n = (f \star g)(n)$ for $n = 1, ..., 5$. The input vector $\mathbf{x}$ is *padded* with extra zeros to make it compatible with the filter length. The output of the convolutional layer $\mathbf{y}$ is called *feature map*. In general, when a 1D convolutional layer applies padding of length p (number of zeros added to each boundary) to an input $\mathbf{x} \in \mathbb{R}^n$ and uses a stride length s, the size of the output is

$$o = \lfloor \frac{n + 2p - k}{s} \rfloor + 1.$$

3.3.5.2 Causal Convolutional Layer

When CNNs are used for time series forecasting, which is a type of sequence modeling, which we introduce in Section 3.4, usually multiple layers of *causal convolutions* are applied. Causal convolutions use convolutional filters designed to ensure that only past information is used for forecasting. A CNN that is used for sequence prediction is referred to as *Temporal Convolutional Network* (TCN). Formally, for a univariate time series $\mathbf{x} = (x_1, ..., x_T)$ and a learnable filter $f : \{0, ..., k-1\} \to \mathbb{R}$, the output at

time step t of a 1D causal convolutional layer is given by

$$y_t = (f \star \mathbf{x})(t) = \sum_{i=0}^{k-1} f(i) \cdot x_{t-i}. \tag{3.29}$$

This means that causal convolutions compute weighted moving averages that only take inputs before t into account. The causal convolution is *translation-equivariant*. In general, a function g is said to be translation-equivariant if

$$g(T(x)) = T(g(x)), \tag{3.30}$$

where T denotes a translation operator. This property enables the causal convolution to detect distinct features at different time steps while maintaining their relative positions in the output, thereby generating a chronological representation.

In Equation 3.29, the *receptive field* of the causal convolution is given by the filter size k. A simple causal convolution is only able to look back at a history with size linear in the depth of the network. A straightforward approach that can exponentially enlarge the receptive field is the use of *dilated causal convolution* layers, which additionally introduce a *dilation factor* d:

$$y_t = (f \star_d \mathbf{x})(t) = \sum_{i=0}^{k-1} f(i) \cdot x_{t-d \cdot i}. \tag{3.31}$$

The dilation factor d is commonly increased exponentially with the depth of the TCN, i.e. $d = O(2^i)$ at layer i of the network [104].

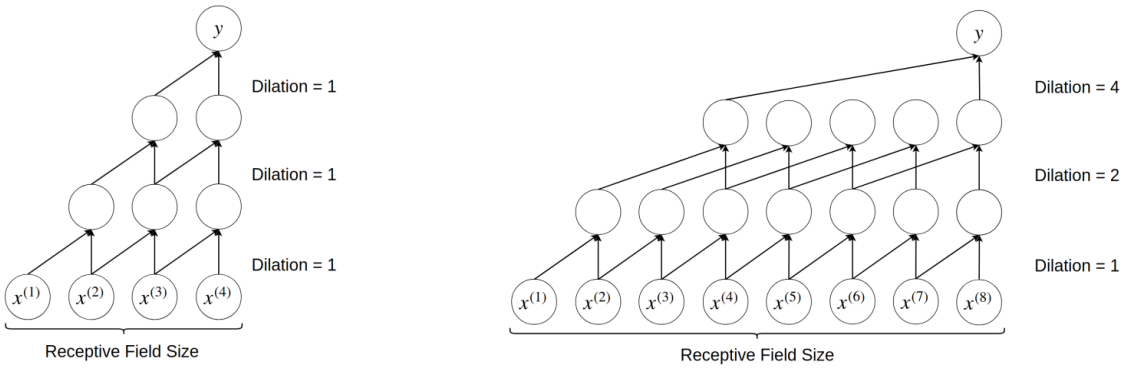


Figure 3.5: Receptive field size for causal and dilated causal convolutions, using the same number of layers [105].

Figure 3.5 illustrates the general structure of a TCN with dilated causal convolutions. Dilated convolutions are particularly useful in time-series applications where long-term dependencies play a crucial role. They can be interpreted as convolutions of a down-sampled version of the lower layer features, effectively reducing resolution to incorporate information from the distant past [95].

3.3.5.3 Pooling Layer

Pooling is used to reduce the dimensionality of a given feature map while highlighting the prominent features, thus enabling the CNN/TCN to learn a global representation of the training data.

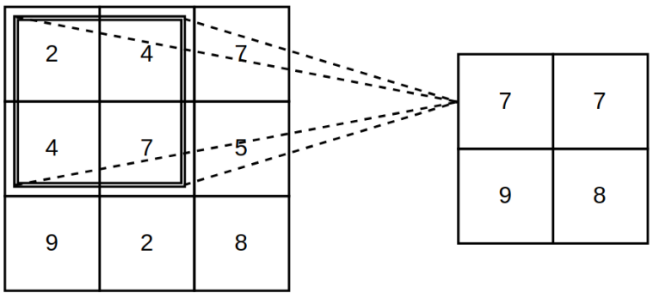


Figure 3.6: Maximum pooling with a 2x2 kernel and stride one [105].

Pooling layers consist of a fixed-shape kernel that is slid over all regions of the input according to some fixed stride length, computing a single output for each location traversed by the kernel, thus providing a summary statistic of nearby outputs of a given feature map. Pooling operators are deterministic; they do not contain any learnable parameters. The most commonly used types are maximum and average pooling operators [106].

3.3.5.4 Flatten Layer

The application of convolutional layers to input data results in multichannel output data, as one channel is produced from each convolutional kernel. The pooling layer processes each channel separately, resulting in the same number of output channels of the pooling layer as it possesses input channels. In order for the output of the convolutional and consecutive pooling layers to be suitable as input for fully connected layers, a flatten layer is required. The flatten layer simply concatenates its multichannel input to form one flat structure which can be fed into a fully connected layer.

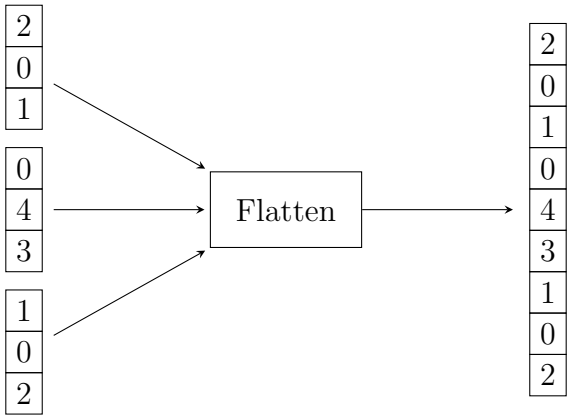


Figure 3.7: 1D Flatten layer.

3.3.6 Regularization Techniques

As described in Section 3.1, regularization refers to any modification of a supervised machine learning algorithm that aims to reduce its generalization error, but not its training error. Finding effective regularization strategies for deep learning models is itself a field of research and regularization can take many forms. In this section, regularization by penalizing model complexity will be discussed.

There is no universal way to define model complexity [107], but complexity measures for specific model classes can be defined based on heuristics. Choosing a model class $\mathcal{F}$ restricts the form of a possible solution and therefore already represents a form of regularization in itself. For a model class $\mathcal{F}$, we can define the complexity measure

$$\Omega : \mathcal{F} \to \mathbb{R}_+.$$

For example, common complexity measures for linear models are based on norms of the coefficient vector $\boldsymbol{\theta}$:

$$\Omega(f) = \begin{cases} \|\boldsymbol{\theta}\|^2 \\ \|\boldsymbol{\theta}\|_1 = \sum_i |\theta_i|. \end{cases}$$

These measures are also used by many models defined through basis function expansions, such as DFNs. In this model class, typically only the weights are regularized as bias terms require a low amount of data to fit accurately and regularizing them can lead to significant underfitting [61]. The complexity measures most commonly used

for neural network regularization are

$$\Omega(f) = \begin{cases} \|\mathbf{w}\|^2 & (L_2 \text{ or Ridge regularization}), \\ \|\mathbf{w}\|_1 & (L_1 \text{ or LASSO regularization}), \\ \lambda_1\|\mathbf{w}\|_1 + \lambda_2\|\mathbf{w}\|^2 & (\text{Elastic net regularization}). \end{cases}$$

In order to control model complexity during training, a *penalized* ERM criterion is introduced:

$$J(f) = \hat{R}(f) + \alpha\Omega(f),$$

where $\alpha \in [0, \infty)$ is a hyperparamter (see Section 3.3.7) controlling how heavily complexity is penalized. This technique is referred to as *weight decay* or *parameter norm* penalization. Penalization can be seen as a flexible method of regularization that enables a model to fit more complex patterns, but only if the training data provides sufficient evidence to support such complexity [58].

Although restricting the model class $\mathcal{F}$ seems to represent a different approach from weight decay regularization, the two techniques turn out to be closely related. In fact, if we view Ω and $\hat{R}$ as convex functions of the model's parameter configuration $\boldsymbol{\theta}$, weight decay is equivalent to restricting $\mathcal{F}$ to the smaller model class

$$\mathcal{F}_\lambda = \{f_{\boldsymbol{\theta}} : \Omega(\boldsymbol{\theta}) < \lambda\},$$

as the following Theorem shows [24].

Theorem 1. *Assume that $\hat{R}(\boldsymbol{\theta})$ and $\Omega(\boldsymbol{\theta})$ are smooth. Then for every local minimum θ^* of*

$$\hat{R}(\boldsymbol{\theta}) \quad s.t. \quad \Omega(\boldsymbol{\theta}) \leq \lambda, \tag{3.32}$$

there exists a λ' such that $\boldsymbol{\theta}^*$ is a local minimum of

$$\hat{R}(\boldsymbol{\theta}) + \lambda' \Omega(\boldsymbol{\theta}). \tag{3.33}$$

Proof. Let $\boldsymbol{\theta}^*$ be a local minimum of (3.32) and let λ be given. Then we either have

$$\Omega(\boldsymbol{\theta}^*) < \lambda \quad \text{or} \quad \Omega(\boldsymbol{\theta}^*) = \lambda.$$

If $\Omega(\boldsymbol{\theta}^*) < \lambda$, we choose $\lambda' = 0$, which makes $\boldsymbol{\theta}^*$ a local minimum of (3.33). For $\Omega(\boldsymbol{\theta}^*) = \lambda$, consider the Lagrange function

$$L(\boldsymbol{\theta}, \eta) = \hat{R}(\boldsymbol{\theta}) + \eta(\Omega(\boldsymbol{\theta}) - \lambda)$$

for the problem

$$\begin{aligned} \min. \quad & \hat{R}(\boldsymbol{\theta}) \\ \text{s.t.} \quad & \Omega(\boldsymbol{\theta}) = \lambda. \end{aligned} \tag{3.34}$$

If $\boldsymbol{\theta}^*$ is a local minimum of (3.34), then a property of Lagrange functions states that there exists η^* such that

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*, \eta^*) = 0,$$

which implies that

$$\nabla_{\boldsymbol{\theta}} \hat{R}(\boldsymbol{\theta}^*) + \eta^* \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}^*) = 0,$$

so $\boldsymbol{\theta}^*$ is also a local minimum of the regularized objective

$$\hat{R}(\boldsymbol{\theta}) + \eta^* \Omega(\boldsymbol{\theta}).$$

□

If $\hat{R}(\boldsymbol{\theta})$ and $\Omega(\boldsymbol{\theta})$ are convex, then every local minimum is a global minimum, and as a result, weight decay and model class restriction are equivalent up to the choice of λ. The equivalence does not hold in the presence of multiple local minima.

3.3.6.1 Dropout

Dropout layers offer a powerful and computationally efficient regularization method for neural networks [108]. They function by randomly setting input units to 0 at a frequency of $p \in (0, 1)$ during each training step, with inputs not set to 0 being scaled by $\frac{1}{1-p}$ to maintain the sum of all inputs. The dropout rate p is a model hyperparameter.

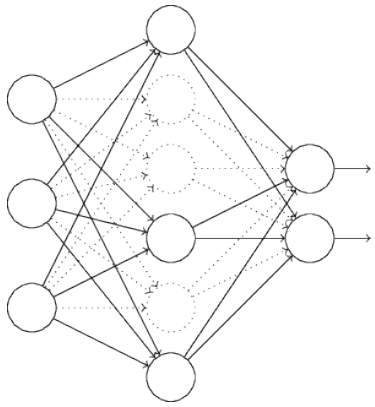


Figure 3.8: Dropout layer in a DFN [109].

Dropout approximates the training and evaluation of an ensemble with exponentially many sub-networks created by excluding non-output units from the base network. For a hidden layer l with n_l neurons, a neural network can generate 2^{n_l} possible thinned networks. Dropout samples from these 2^{n_l} networks and trains the sampled models, using a randomly sampled binary mask for each dropout layer to set respective inputs to zero during weight update cycles [61]. For prediction, an unthinned network with adjusted weights based on the induced probability law from random drops is used.

By injecting noise into the learning process, dropout encourages the model to learn more robust representations, compelling the remaining active neurons to adapt and share the responsibility of making predictions. This results in a more regularized model that is less prone to overfitting.

3.3.6.2 Early Stopping

Early stopping is a widely-used regularization technique for training neural networks that aims to prevent overfitting by halting the training process at an optimal point. The method involves monitoring a performance metric, such as validation loss, during the training process and stopping the training when the monitored metric fails to improve for a specified number of consecutive iterations, known as the *patience* parameter, which is a model hyperparameter.

The rationale behind early stopping is that as training progresses, the model tends to fit the training data more closely, reducing the training error. However, after a certain point, the model starts to capture the noise present in the training data, causing the validation error to increase, which indicates overfitting. Model regularization is achieved by stopping training early, before the model becomes overly specialized to the training data.

This technique can be considered an implicit form of model selection, as it chooses the best-performing model during the training process without explicit hyperparameter tuning. It is computationally efficient, as it does not require the training of multiple models, and can be combined with other regularization techniques [61].

3.3.7 Hyperparameters

Machine learning model parameters that control the behavior of the learning process are denoted as hyperparameters. Unlike the model's learnable parameters, hyperparameters are not adjusted by the machine learning algorithm itself, as inferring hyperparameters from training data would always result in overfitted models [61]. Hyperparameter selection can be guided by a procedure called *cross-validation*. Before assessing the model's performance on a test set, its generalization error can be estimated during or after learning by using a validation set of examples that the training algorithm does not observe. Hyperparameters can then be updated in a way that minimizes the estimated generalization error. As samples from the test set cannot be used to make choices about model hyperparameters, validation sets need to be constructed from training data. In a procedure called k-fold cross-validation, the training dataset is split into k non-overlapping subsets. The generalization error is estimated by taking the average test error across k trials.

The most common strategies for splitting training data assume that the examples are independent and identically distributed, which does not hold for time series data due to the temporal dependencies between observations. Performing cross-validation when training machine learning models for time series forecasting is not trivial as the model should never be presented with future observations during training. The temporal dependency between observations needs to be preserved during training and testing. Currently, there is no consensus on the most suitable method for estimating model performance in time series forecasting tasks. Cerqueira et al. [110] conducted experiments on 53 real-world time series with potential non-stationarities and three

stationary synthetic time series, ultimately concluding that *out-of-sample methods* are the most accurate in real-world scenarios. These methods have traditionally been used to estimate predictive performance on time-dependent data by reserving a portion of the time series for testing. Although these methods do not leverage the entire dataset, they preserve the temporal order of observations, which is important for controlling the dependency among observations and accounting for the temporal evolution of the time series.

3.4 Recurrent Neural Networks

While time series forecasting can be formulated as a problem suitable for arbitrary algorithms capable of supervised learning, in this section we introduce a model that was explicitly designed to handle sequential data, the *Recurrent Neural Network* (RNN) [111].

Time series forecasting is a type of *sequence modeling task*. Formally, a sequence modeling task is given an input sequence $\{x_t\}_{t=1}^{T}$ and aims to predict the corresponding outputs $\{y_t\}_{t=1}^{T}$ for each time step t. Crucially, the output y_t at time t can only rely on previously observed inputs $x_1, ..., x_t$ and not on any subsequent inputs $x_{t+1}, ...x_T$. A sequence modeling network [112] is any function

$$f : \mathcal{X}^T \to \mathcal{Y}^T$$

which adheres to this causality constraint and establishes the mapping

$$\hat{y}_1, ..., \hat{y}_T = f(x_1, ..., x_T).$$

From a probabilistic perspective, the problem of predicting the next value of a sequence conditioned on all previous timesteps can be modeled by maximizing the likelihood of the target value, given all previous steps [105]. Given a predictor parametrized by $\boldsymbol{\theta}$, the task is to maximize y_t conditioned on all previous steps.

$$p(y|x) = \prod_{t=1}^{T} p(y_t|x_1, ..., x_T; \boldsymbol{\theta}) \tag{3.35}$$

Recurrent neural networks are parametric models that estimate the conditional distribution in Equation 3.35 by introducing a latent variable, called *hidden state* h_t, that encodes the sequence information up to time step t.

An RNN can be viewed as an *encoder-decoder* architecture that stores historical information into the latent variable and computes the prediction using only h_t:

$$\hat{y}_t = g_{dec}(h_t), \tag{3.36}$$

$$h_t = g_{enc}(x_1, ..., x_t). \tag{3.37}$$

The encoding of historical information in RNNs is accomplished by incorporating cycles into their underlying network topology, functioning as feedback loops. During backpropagation, the network is "unfolded in time," as depicted in Figure 3.9.

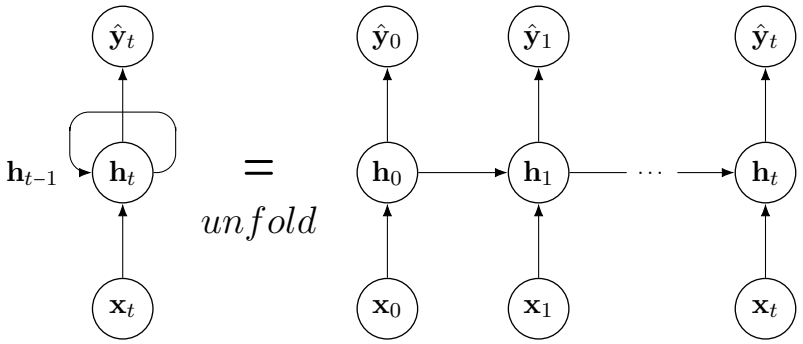


Figure 3.9: Folded and unfolded simple Recurrent Neural Network.

In general, RNNs are able to model one-to-one, one-to-many, many-to-one or many-to-many relations. For simplicity, the subsequent discussions on RNN training will focus on the time-synchronous many-to-many scenario, as previously introduced, and we let $\mathcal{X} = \mathbb{R}^n$ and $\mathcal{Y} = \mathbb{R}^m$.

3.4.1 Forward Pass

The standard RNN is a nonlinear dynamical system that maps sequences to sequences [24]. In the multivariate case, it is given an input sequence $\{\mathbf{x}_t\}_{t=1}^T$ from $\mathbb{R}^n$ and computes a sequence of hidden states $\{\mathbf{h}_t\}_{t=1}^T$ from $\mathbb{R}^s$ and a sequence of outputs $\{\mathbf{y}_t\}_{t=1}^T$ from $\mathbb{R}^m$. The structure of a single cell as used in standard RNNs is showcased in Figure 3.10.

The RNN can be represented as a deep feedforward neural network with T stages of computation, as shown in Figure 3.11. It is parametrized with three weight matrices and three bias vectors that are *shared* across time steps.

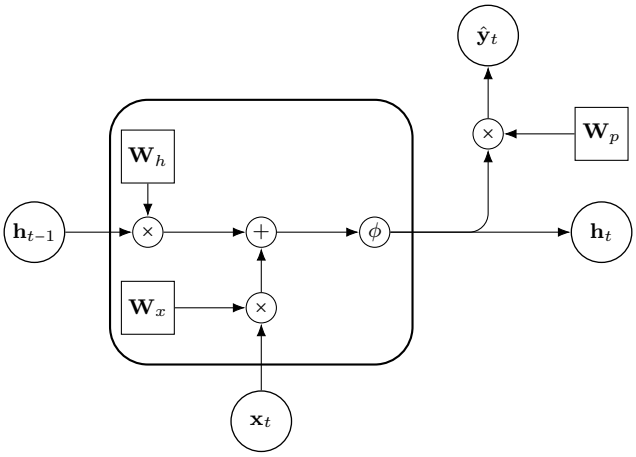


Figure 3.10: Structure of the standard RNN cell.

The computation of the hidden state is governed by the equation

$$\mathbf{h}_t = \phi(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{b}_h + \mathbf{W}_x\mathbf{x}_t + \mathbf{b}_x), \tag{3.38}$$

where ϕ is a nonlinear activation function that is applied elementwise. The network's output $\hat{\mathbf{y}}_t$ is computed by

$$\hat{\mathbf{y}}_t = \psi(\mathbf{W}_p\mathbf{h}_t + \mathbf{b}_p), \tag{3.39}$$

where ψ is typically a nonlinear elementwise function.

In order to compute the first state $\mathbf{h}_1$, we need an initial state $\mathbf{h}_0$, which is commonly set to zero [113]. $\mathbf{W}_h$ is the recurrent matrix that carries the translative effect of previous time steps. By expressing $\mathbf{h}_t$ as a parametrized function of $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$, it is easy to see after recursive substitution:

$$\mathbf{h}_t = \phi(\mathbf{x}_t, \mathbf{h}_{t-1}; \boldsymbol{\theta}) = \phi(\mathbf{x}_t, \phi(\mathbf{x}_{t-1}, \phi(...\phi(\mathbf{x}_1, \mathbf{h}_0; \boldsymbol{\theta})...; \boldsymbol{\theta}); \boldsymbol{\theta}).$$

This is means that $\mathbf{h}_t$ contains information about the entire sequence $\mathbf{x}_1, ..., \mathbf{x}_t$, analogous to Equation 3.36. This *memory* will usually be lossy since an entire sequence of variable length is mapped to a single vector.

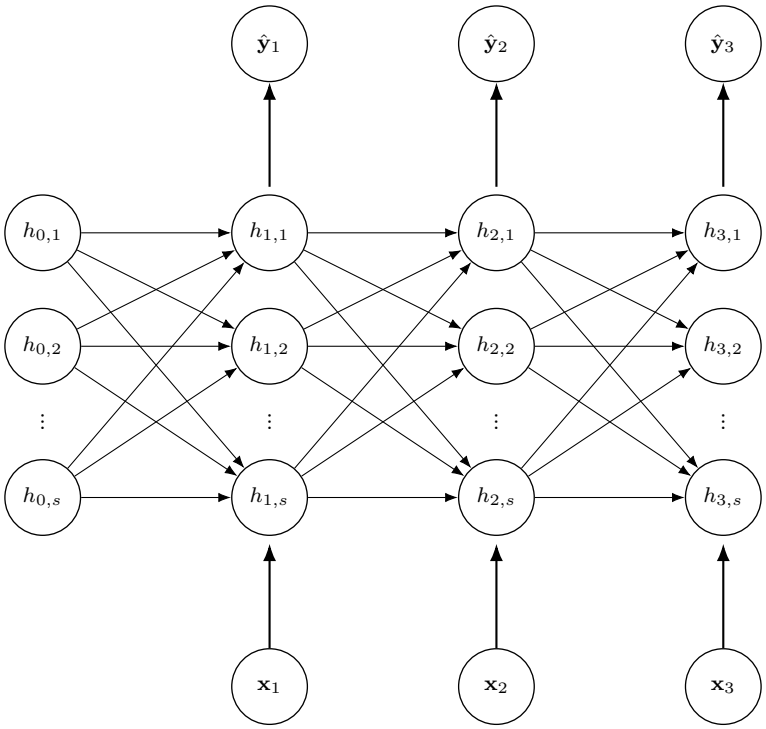


Figure 3.11: RNN represented as a DFN with $T = 3$ stages of computation.

In the time-synchronous output setting, the network loss is calculated as a sum of per-timestep losses. If we let $f : \mathbb{R}^n \to \mathbb{R}^m$ denote the RNN model, the empirical risk of the network's parameter configuration $\boldsymbol{\theta}$ for a sequential training dataset $\mathcal{D} = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{T}$ of finite length T is given by

$$\begin{aligned} \hat{R}(\boldsymbol{\theta}) &= \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}(f(\mathbf{x}_t;\boldsymbol{\theta}),\mathbf{y}_t) \\ &= \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}_t(\boldsymbol{\theta}), \end{aligned} \tag{3.40}$$

where $\mathcal{L}$ is a differentiable loss function and $\mathcal{L}_t$ computes the loss incurred by the model's output at time t and is defined analogously to Equation 3.20.

3.4.2 Backpropagation Through Time

In RNNs, the backpropagation of gradients is commonly referred to as *backpropagation through time* (BPTT) [114]. During BPTT, gradients are computed for each time step and then backpropagated through the unfolded network, as depicted in Figure 3.12.

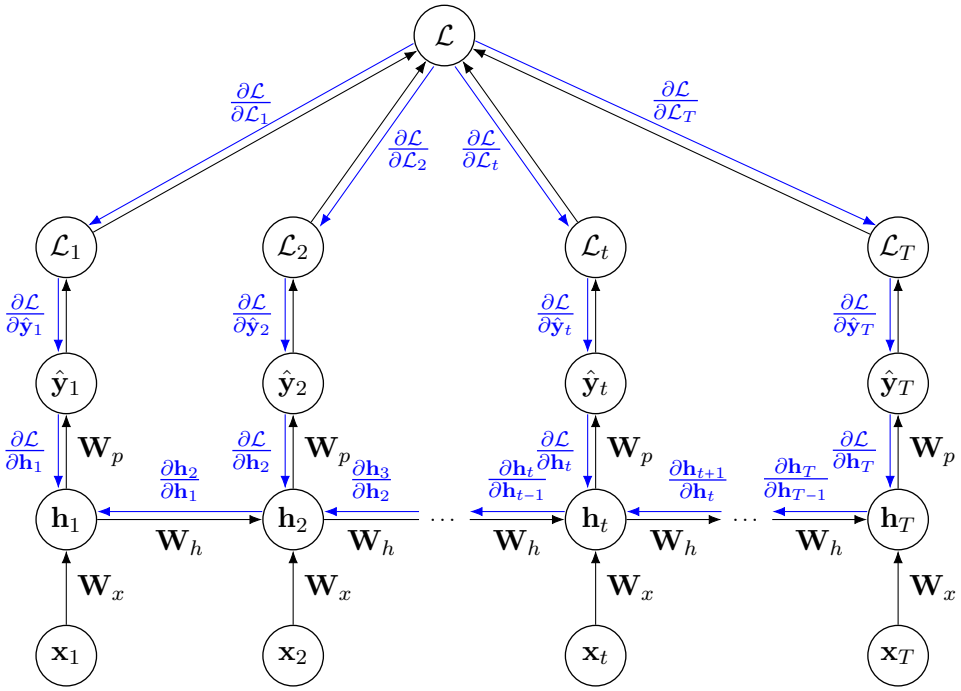


Figure 3.12: Gradient flow through an unfolded RNN.

When considering backpropagation of gradients through RNNs, it is crucial to note that each weight is present at every stage of the unfolded network. Until now, only the case of unique weights in neural networks was considered. However, it is possible to transform any network that has duplicated weights into a network that has unique weights [83]. Assume that after the feed-forward step, the state of the network is the one shown in Figure 3.13. In the original network, weight w is replicated and assigned different inputs at two different points. However, the transformed network shown in Figure 3.14 produces the same results as the original network.

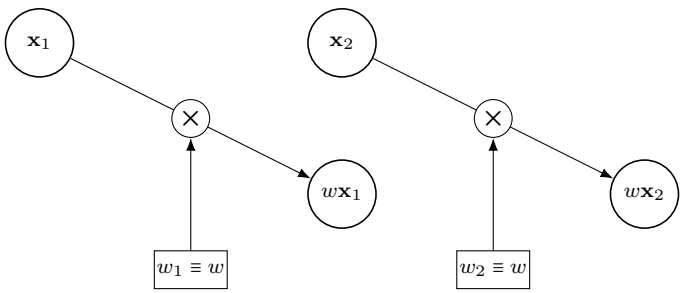


Figure 3.13: A duplicated weight in a network.

The transformed network has two groups of paths, each originating from one of the two multipliers connected to node w. Backpropagation can be performed as usual in

the original network and since w_1 is the same variable as w_2, we get

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial w_1} + \frac{\partial \mathcal{L}}{\partial w_2}.$$

In general, when the same weight is associated with multiple edges in a neural network, backpropagation is performed separately for each edge, as is done for weights that are not shared, e.g. in FFNs. Once the gradients have been calculated for each edge, they are simply added to obtain the final gradient for the shared weight. This is analogous to viewing $w^{(t)}$ as a copy of w which is only used at time step t [115].

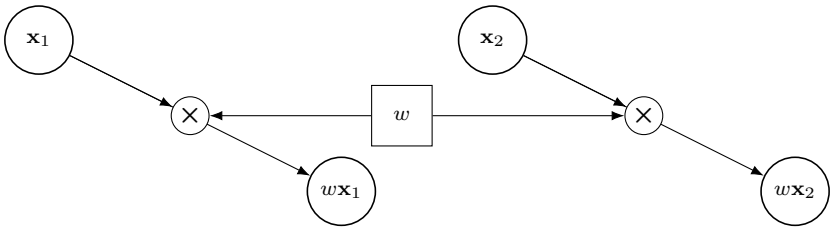


Figure 3.14: Transformed network.

For simplicity, in the following derivation of parameter update rules based on backpropagation through time (BPTT), we assume an RNN using squared error loss function $\mathcal{L}$, hyperbolic tangent as hidden nonlinearity ϕ, and the identity function as output activation ψ.

At time step t, applying the chain rule to Equation 3.40 allows us to write $\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t}$ as

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} = \hat{\mathbf{y}}_t - \mathbf{y}_t.$$

At the final time step T, the only descendant of $\mathbf{h}_T$ is $\hat{\mathbf{y}}_T$, so we derive the error gradient with respect to the hidden state from Equation 3.39 as

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} &= \mathbf{W}_p^\top \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_T} \\ &= \mathbf{W}_p^\top (\hat{\mathbf{y}}_T - \mathbf{y}_T). \end{aligned}$$

The gradients can now be propagated backward through time from $t = T-1$ to $t = 1$. It is important to keep in mind that every hidden state $\mathbf{h}_t$ has $\mathbf{h}_{t+1}$ and $\hat{\mathbf{y}}_t$ as descendants. By considering the properties of the hyperbolic tangent, the error gradient with respect

to the hidden state $\mathbf{h}_t$ is thus given by

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} &= \left(\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} + \left(\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t} \\ &= \mathbf{W}_h^\top \left(\left(\mathbf{1} - \mathbf{h}_{t+1}^2\right) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}}\right) + \mathbf{W}_p^\top \left(\hat{\mathbf{y}}_t - \mathbf{y}_t\right).\end{aligned}$$

In order to obtain the gradients with respect to the RNN's parameters, we will first consider the contribution of a single parameter to the whole gradient. The final gradients can then be obtained by adding the contributions of all instantiations of an edge in time:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \theta^{(t)}}.$$

This procedure yields the following update rules for the RNN's output parameters:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p} = \sum_{t=1}^{T} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial (\hat{\mathbf{y}}_t)_i} \frac{\partial (\hat{\mathbf{y}}_t)_i}{\partial \mathbf{W}_p^{(t)}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t} \mathbf{h}_t^\top = \sum_{t=1}^{T} \left(\hat{\mathbf{y}}_t - \mathbf{y}_t\right) \mathbf{h}_t^\top, \tag{3.41}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_p} = \sum_{t=1}^{T} \left(\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{b}_p^{(t)}}\right)^\top \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t} = \sum_{t=1}^{T} \left(\hat{\mathbf{y}}_t - \mathbf{y}_t\right), \tag{3.42}$$

The gradients with respect to the cell parameters are derived analogously:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \sum_{t=1}^{T} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial (\mathbf{h}_t)_i} \frac{\partial (\mathbf{h}_t)_i}{\partial \mathbf{W}_h^{(t)}} = \sum_{t=1}^{T} \left(\left(\mathbf{1} - \mathbf{h}_t^2\right) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}\right) \mathbf{h}_{t-1}^\top, \tag{3.43}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_h} = \sum_{t=1}^{T} \left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_h^{(t)}}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \sum_{t=1}^{T} \left(\mathbf{1} - \mathbf{h}_t^2\right) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}, \tag{3.44}$$

(3.45)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_x} = \sum_{t=1}^{T} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial (\mathbf{h}_t)_i} \frac{\partial (\mathbf{h}_t)_i}{\partial \mathbf{W}_x^{(t)}} = \sum_{t=1}^{T} \left(\left(\mathbf{1} - \mathbf{h}_t^2\right) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}\right) \mathbf{x}_t^\top, \tag{3.46}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_x} = \frac{\partial \mathcal{L}}{\partial \mathbf{b}_h}. \tag{3.47}$$

The weight update rules that are obtained after forward pass and BPTT in an RNN can be used by a suitable gradient-based optimization algorithm to minimize the network's empirical risk as described in Section 3.3.3. Although the gradients are straightforward to compute, RNNs are fundamentally hard to train, especially on problems with long-term temporal dependencies [75]. We examine this issue in the next section.

3.4.2.1 The Vanishing or Exploding Gradient Problem

The distance between a prediction and the last relevant input can be large, requiring the relevant information to be held in the hidden state for a large number of time steps. This concept is known as long-term dependencies, where the computation of the output at time t relies on the input from a significantly earlier time $\tau \ll t$ [75].

From Equation 3.40 and the fact that the network parameters are shared across time, it is easy to see that

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \theta}$$

for some trainable network parameter $\theta \in \mathbb{R}$ that is involved in hidden state updates. If we let the term $\frac{\partial \mathbf{h}_k}{\partial \theta}$ denote the partial derivative of $\mathbf{h}_k$ where $\mathbf{h}_{k-1}$ is taken as a constant with respect to θ, we can derive the following expression for $\frac{\partial \mathcal{L}_t}{\partial \theta}$ from Equations 3.38 and 3.40:

$$\frac{\partial \mathcal{L}_t}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \theta},$$

where

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = \prod_{i=k+1}^{t} \mathbf{W}_h^\top diag(\phi'(\mathbf{h}_{i-1})).$$

diag converts a vector into a diagonal matrix and ϕ' is the elementwise applied derivative of ϕ. This means that backpropagating the gradient over l time steps requires computing the product of l Jacobian matrices, which causes the error gradient to either vanish or explode over time, as shown in the following Theorem [75]:

Theorem 2. *If $\mathbf{W}_h$ is diagonalizable and ϕ is the identity function, the term*

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$$

grows exponentially fast with t .

Proof. Assume that $t \to \infty$ and $l = t - k$. It is easy to see from Equation 3.38 that

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \left(\mathbf{W}_h{}^\top\right)^l. \tag{3.48}$$

Let $\mathbf{W}_h$ have eigenvalues $\lambda_1, ..., \lambda_s$ with $|\lambda_1| > |\lambda_2| > ... > |\lambda_s|$ and corresponding eigen-

vectors $\mathbf{p}_1, ..., \mathbf{p}_s$ which form an eigenbasis. We can now express $\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}$ using this basis:

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} = \sum_{i=1}^{s} c_i \mathbf{p}_i^\top.$$

By applying power iteration, we get $\mathbf{p}_i^\top (\mathbf{W}_h^\top)^l = \lambda_i^l \mathbf{p}_i^\top$ for $l \to \infty$. If $c_j \neq 0$ and $c_{j'} = 0$ for any $j' < j$, we get

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} &= c_j \lambda_j^l \mathbf{p}_j^\top + \sum_{i=j+1}^{s} c_i \lambda_i^l \mathbf{p}_i^\top \\ &= c_j \lambda_j^l \mathbf{p}_j^\top + \lambda_j^l \sum_{i=j+1}^{s} c_i \frac{\lambda_i^l}{\lambda_j^l} \lambda_i^l \mathbf{p}_i.\end{aligned}$$

Because $\left|\frac{\lambda_i}{\lambda_j}\right| < 1$ for $i > j$, we have $\lim_{l\to\infty} \left|\frac{\lambda_i}{\lambda_j}\right|^l = 0$. If $|\lambda_j| > 1$, it follows that the term $\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ grows exponentially with l in the direction $\mathbf{p}_j$. □

This provides a necessary condition for gradients to grow, namely $\|\mathbf{W}_h\|_2 > 1$. By inverting the proof, it can be shown that if the largest eigenvalue of $\mathbf{W}_h$ is smaller than 1, the gradients will vanish for $t \to \infty$, which provides a sufficient condition for vanishing gradients. The proof can be extended to non-diagonalizable matrices by using the Jordan normal form of $\mathbf{W}_h$.

Theorem 2 can be generalized for nonlinear functions ϕ with $\|diag(\phi'(\mathbf{h}_k))\|_2 \leq \gamma$ for a constant $\gamma \in \mathbb{R}$.

Theorem 3. *Given a nonlinear activation function ϕ with the above property and the weight matrix $\mathbf{W}_h$ with $\|\mathbf{W}_h\| < \frac{1}{\gamma}$, the term $\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ vanishes exponentially fast with t.*

Proof. The matrix $\frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k}$ is given by $\mathbf{W}_h^\top diag(\phi'(\mathbf{h}_k))$. Due to our assumptions, we have

$$\begin{aligned}\left\|\frac{\partial \mathbf{h}_{k+1}}{\mathbf{h}_k}\right\|_2 &= \|\mathbf{W}_h^\top diag(\phi'(\mathbf{h}_k))\|_2 \\ &\leq \|\mathbf{W}_h^\top\|_2 \, \|diag(\phi'(\mathbf{h}_k))\|_2 < \frac{1}{\gamma}\gamma < 1 \quad \text{for all } k.\end{aligned} \tag{3.49}$$

Let $\eta \in \mathbb{R}$ be such that $\left\|\frac{\partial \mathbf{h}_{k+1}}{\mathbf{h}_k}\right\|_2 \leq \eta < 1$ for all k. Such an η exists due to Equation 3.49. We can now show by induction over i that

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}\prod_{i=k}^{t-1}\frac{\partial \mathbf{h}_{i+1}}{\partial \mathbf{h}_i} \leq \eta^{t-k}\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}. \tag{3.50}$$

Because we have $\eta < 1$, it follows that $\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ vanishes exponentially fast with t. □

The proof can be inverted to obtain the necessary condition for exploding gradients, namely $\|\mathbf{W}_h\|_2 > \frac{1}{\gamma}$. For *tanh* we have $\gamma = 1$ and for *sigmoid* we have $\gamma = \frac{1}{4}$.

When training a standard RNN, the problem of vanishing error gradients primarily affects earlier time steps. When this issue arises at the very beginning of the network, the entire model's accuracy becomes compromised [116]. Conversely, exploding gradients cause the weights to oscillate and push the network into a chaotic regime, preventing gradient descent from converging [117].

One approach that addresses both vanishing and exploding gradients is limiting the number of time steps of error backpropagation. This method is referred to as truncated BPTT and can improve the learning process in standard RNNs [118].

Aside from special weight initialization and batch normalization techniques, the most common approach to address vanishing gradients is using advanced RNN architectures with gating mechanisms. These architectures are discussed in detail in the subsequent sections.

Using an L_1- or L_2-penalty on the weights can help with exploding gradients, but at the same time imposes a rigid regime on the recurrent network, limiting its versatility and long-term memory capabilities [75].

A widely used and straightforward method for preventing exploding gradients is *gradient clipping*.

3.4.2.2 Gradient Clipping

An easy-to-implement and computationally efficient mechanism to deal with sudden increases in the norm of error gradients is to rescale them whenever their norm exceeds a certain threshold. When minimizing a differentiable loss function $\mathcal{L}$ during neural network training via gradient descent with gradient clipping, we have the following update rules:

$$
\begin{aligned}
\mathbf{g_n} &= \nabla \mathcal{L}(\boldsymbol{\theta}^{(n-1)}), \\
\mathbf{g_n} &= \begin{cases} \mathbf{g}_n, & \text{if } \|\mathbf{g}_n\|_2 < \xi \\ \frac{\xi}{\|\mathbf{g}_n\|_2}, & \text{if } \|\mathbf{g}_n\|_2 \geq \xi, \end{cases} \\
\boldsymbol{\theta}_n &= -\gamma_n \mathbf{g}_n, \\
\boldsymbol{\theta}^{(n)} &= \boldsymbol{\theta}^{(n-1)} + \boldsymbol{\theta}_n.
\end{aligned}
$$

The *threshold* $\xi \in \mathbb{R}_+$ is an additional hyperparameter. The algorithm can be thought of as an adaptation of the learning rate based on the norm of the gradient [116].

The authors in [101] state that using gradient clipping can accelerate empirical convergence when applied during neural network training.

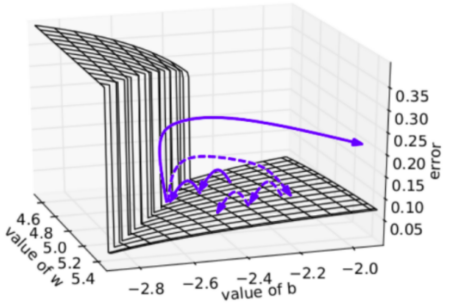


Figure 3.15: Error surface of a single hidden unit RNN exhibiting high curvature walls. Solid lines are standard trajectories gradient descent might follow, dashed lines are trajectories when clipping is applied [116].

3.5 Long Short-Term Memory

The Long Short-Term Memory (LSTM) architecture, first introduced by Hochreiter and Schmidhuber in [119], was designed to address the problem of handling long-term dependencies in sequential data. While traditional RNNs struggle with retaining information over long periods of time, LSTM networks use additional structures that allow them to selectively remember and forget information. This approach is inspired by the way the human brain processes sequential information, where we forget redundant parts of the input in order to retain relevant information.

LSTM networks have become a popular choice for a variety of tasks, including machine translation [120], speech recognition [121], robot control [122], and time series forecasting [84]. In recent years, LSTM networks have achieved unprecedented success and are widely regarded as a significant step towards Artificial General Intelligence [123, 124, 125]. Regarding EV energy demand forecasting, LSTM models have shown promising results in [15] and [126]. They are also capable of accurately predicting EV arrival and departure times at charging stations, as shown in [56].

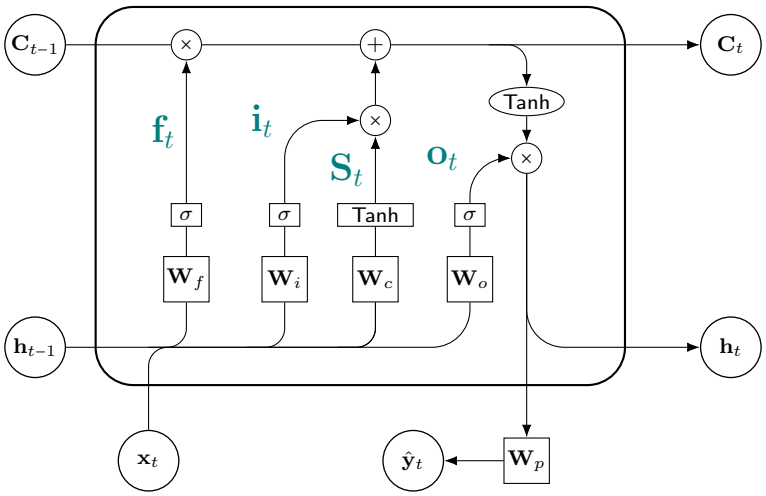


Figure 3.16: Structure of the LSTM cell.

LSTM networks are specifically designed to address the problem of vanishing gradients in standard RNNs. The basic idea behind LSTM is to create paths through time where gradients can flow without vanishing. This is achieved by introducing a *memory cell*, which has the ability to add or remove contextually relevant information. The *cell state* $\mathbf{C}_t$ represents the memory cell at time t. The memory cell is the fundamental recurrent neural unit in an LSTM network and is regulated by gates, which are controlled by learned weights and biases. The structure of an LSTM cell is showcased in Figure 3.16. The key mechanism of LSTMs is the horizontal line in the upper part of the cell through which the cell states are transmitted. It traverses the network and is only involved in linear interactions. This absence of nonlinearities along the cell state's transmission path addresses the problem of gradient instabilities.

As variant of the standard RNN, LSTM networks are trained analogously via forward pass, BPTT and parameter update using a suitable gradient-based optimization algorithm.

3.5.1 Forward Pass

We once again consider the time-synchronous output setting. Like the standard RNN, an LSTM network is a nonlinear dynamical system that maps an input sequence $\{\mathbf{x}_t\}_{t=1}^T$ from $\mathbb{R}^n$ to an output sequence $\{\hat{\mathbf{y}}_t\}_{t=1}^T$ from $\mathbb{R}^m$ while recording a sequence of hidden states $\{\mathbf{h}_t\}_{t=1}^T$ from $\mathbb{R}^s$ and a sequence of cell states $\{\mathbf{C}_t\}_{t=1}^T$ from $\mathbb{R}^s$. At each time step t, the update of $\mathbf{h}_t$ is governed by

$$\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{C}_t). \tag{3.51}$$

The cell state $\mathbf{C}_t$ is computed via

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \mathbf{S}_t, \tag{3.52}$$

where

$$\mathbf{S}_t = tanh(\mathbf{W}_{hc} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{xc} \cdot \mathbf{x}_t + \mathbf{b}_c)$$

is called *candidate state* and $\mathbf{f}_t$, $\mathbf{i}_t$ and $\mathbf{o}_t$ are called *forget*, *input* and *output gate*, respectively. Their evolution is governed by

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_{hf} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{xf} \cdot \mathbf{x}_t + \mathbf{b}_f), \\
\mathbf{i}_t &= \sigma(\mathbf{W}_{hi} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{xi} \cdot \mathbf{x}_t + \mathbf{b}_i), \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{ho} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{xo} \cdot \mathbf{x}_t + \mathbf{b}_o),
\end{aligned}$$

and $\mathbf{W}, \mathbf{b}$ are learnable weights and biases for each component, respectively, and σ is a *sigmoid* function.

The cell state $\mathbf{C}_t$ equips the LSTM network with an extended memory of past events. Vanishing gradients are mitigated through a *Constant Error Carousel* (CEC), discussed in Section 3.5.2.1. The cell state $\mathbf{C}_t$ can be viewed as an encoded memory featuring dynamic read and update operations.

Update operations are controlled by the forget gate $\mathbf{f}_t$, which decides the portions of the previous cell state's values inherited by the current cell state $\mathbf{C}_t$, based on new information $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$. The input gate determines which portions of the candidate state $\mathbf{S}_t$ should be read into the new cell state. The candidate state $\mathbf{S}_t$, generated from the hidden state $\mathbf{h}_{t-1}$ and new information $\mathbf{x}_t$, is analogous to the hidden state $\mathbf{h}_t$ in standard RNNs. However, unlike the latter, it does not directly

transmit information to the next state; instead, it is modified by multiplication with $\mathbf{i}_t$. In the extreme case of $\mathbf{f}_t \approx \mathbf{1}$ and $\mathbf{i}_t \approx \mathbf{0}$, the cell state (almost) only inherits from its previous cell state.

The hidden state $\mathbf{h}_t$ conveys information using a different path than the cell state. It represents the working memory part of LSTM networks and is present in standard RNNs. It carries information from immediately previous events and overwrites at every time step uncontrollably. The output gate $\mathbf{o}_t$ determines the portion of $tanh(\mathbf{C}_t)$ to be conveyed to the next hidden state, thereby protecting future cell states from perturbation by currently irrelevant memory contents.

All four components $\mathbf{S}_t$, $\mathbf{f}_t$, $\mathbf{i}_t$ and $\mathbf{o}_t$ are themselves standard RNNs and might individually suffer from gradient instabilities. This is avoided through their specific coupling given by Equations 3.52 and 3.51. During training of LSTM networks, the gates learn to open and close access to constant error flow [119].

The prediction $\hat{\mathbf{y}}_t$, loss $\mathcal{L}_t$ and overall network loss are computed just like in standard RNNs, as given in Equations 3.39 and 3.40.

3.5.2 Backpropagation Through Time

Analogous to Section 3.4.2, the parameter update rules are derived by introducing variables $\theta^{(t)}$ that are defined as copies of the respective parameters, used only at time step t. The gradients of $\mathcal{L}$ with respect to the output parameters are derived analogously to the respective gradients in standard RNNs: At time step t, applying the chain rule to Equations 3.39 and 3.40 yields

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} = \hat{\mathbf{y}}_t - \mathbf{y}_t.$$

At the final time step T, the only descendant of $\mathbf{h}_T$ is $\hat{\mathbf{y}}_T$. With equation 3.39, we compute the gradient with respect to the hidden state as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} = \mathbf{W}_p^\top \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_T} = \mathbf{W}_p^\top (\hat{\mathbf{y}}_T - \mathbf{y}_T).$$

The only descendant of $\mathbf{C}_T$ is $\mathbf{h}_T$ and we have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{C}_T} &= \left(\frac{\partial \mathbf{h}_T}{\partial \mathbf{C}_T}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} \\ &= \mathbf{o}_T \odot (\mathbf{1} - tanh(\mathbf{C}_T)^2) \odot (\mathbf{W}_p^\top (\hat{\mathbf{y}}_T - \mathbf{y}_T)).\end{aligned}$$

For $0 \leq t < T$, every hidden state $\mathbf{h}_t$ has $\mathbf{S}_{t+1}$, $\mathbf{f}_{t+1}$, $\mathbf{i}_{t+1}$ and $\mathbf{o}_{t+1}$ as descendants and we have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} &= \left(\frac{\partial \mathbf{S}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{S}_{t+1}} + \left(\frac{\partial \mathbf{f}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{f}_{t+1}} + \left(\frac{\partial \mathbf{i}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{i}_{t+1}} + \left(\frac{\partial \mathbf{o}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{o}_{t+1}} \\ &= \mathbf{W}_{hc}^\top \left((\mathbf{1} - \mathbf{S}_{t+1}^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{S}_{t+1}}\right) + \mathbf{W}_{hf}^\top \left(\mathbf{f}_{t+1} \odot (\mathbf{1} - \mathbf{f}_{t+1}) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{f}_{t+1}}\right) \\ &\quad + \mathbf{W}_{hi}^\top \left(\mathbf{i}_{t+1} \odot (\mathbf{1} - \mathbf{i}_{t+1}) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{i}_{t+1}}\right) + \mathbf{W}_{ho}^\top \left(\mathbf{o}_{t+1} \odot (\mathbf{1} - \mathbf{o}_{t+1}) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{o}_{t+1}}\right).\end{aligned}$$

For $0 \leq t < T$, the cell state $\mathbf{C}_t$ has $\mathbf{C}_{t+1}$ and $\mathbf{h}_t$ as descendants, so we get

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} &= \left(\frac{\partial \mathbf{C}_{t+1}}{\partial \mathbf{C}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{C}_{t+1}} + \left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{C}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \\ &= \mathbf{f}_t \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_{t+1}} + \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}.\end{aligned}$$

For $0 < t \leq T$, by examining the respective descendants of $\mathbf{S}_t$, $\mathbf{f}_t$, $\mathbf{i}_t$ and $\mathbf{o}_t$, we derive

$$\frac{\partial \mathcal{L}}{\partial \mathbf{S}_t} = \left(\frac{\partial \mathbf{C}_t}{\partial \mathbf{S}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} = \mathbf{i}_t \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{f}_t} = \left(\frac{\partial \mathbf{C}_t}{\partial \mathbf{f}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} = \mathbf{C}_{t-1} \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{i}_t} = \left(\frac{\partial \mathbf{C}_t}{\partial \mathbf{i}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} = \mathbf{C}_{t-1} \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{o}_t} = \left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = tanh(\mathbf{C}_t) \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t}.$$

The gradients with respect to the LSTM cell's parameters can now be computed by adding the contributions of all instantiations of a parameter $\theta^{(t)}$ in time, analogous to section 3.4.2. The expressions $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_p}$ are identical to Equations 3.41 and 3.42, respectively. The rather lengthy and repetitive derivation of the parameter update rules can be found in Appendix A.

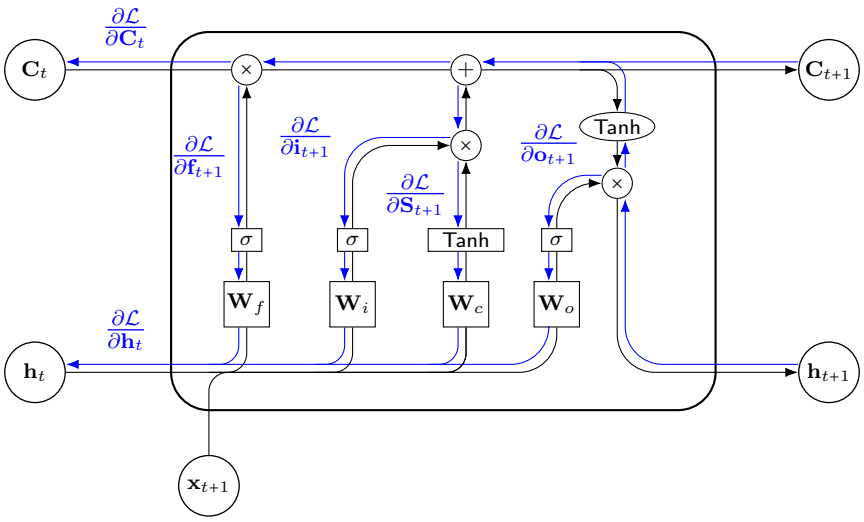


Figure 3.17: Gradient flow through an LSTM cell.

The significance of the horizontal path connecting the cell states in the upper part of the cell in Figure 3.17 is discussed in the subsequent section.

3.5.2.1 Constant Error Carousel

As mentioned before, the cell state $\mathbf{C}_t$ represents a dynamic memory unit whose state is controlled by the forget gate layer and the input gate layer, i.e.

$$\underbrace{\mathbf{memory}_t}_{\mathbf{C}_t} = \underbrace{\mathbf{forget}}_{\mathbf{f}_t} \odot \underbrace{\mathbf{memory}_{t-1}}_{\mathbf{C}_{t-1}} + \underbrace{\mathbf{read}}_{\mathbf{i}_t} \odot \underbrace{\mathbf{features}_t}_{\mathbf{S}_t}.$$

Gradients of the hidden state and the cell state in terms of a parameter θ that is involved in recursive updates can be derived from Equations 3.52 and 3.51 using the chain rule, the product rule and the properties of the hyperbolic tangent function:

$$\frac{\partial \mathbf{h}_t}{\partial \theta} = \mathbf{o}_t \odot (\mathbf{1} - tanh^2(\mathbf{C}_t)) \frac{\partial \mathbf{C}_t}{\partial \theta} + \frac{\partial \mathbf{o}_t}{\partial \theta} \odot tanh(\mathbf{C}_t),$$

$$\frac{\partial \mathbf{C}_t}{\partial \theta} = \frac{\partial \mathbf{f}_t}{\partial \theta} \odot \mathbf{C}_{t-1} + \mathbf{f}_t \odot \frac{\partial \mathbf{C}_{t-1}}{\partial \theta} + \frac{\partial \mathbf{i}_t}{\partial \theta} \odot \mathbf{S}_t + \mathbf{i}_t \odot \frac{\partial \mathbf{S}_t}{\partial \theta}.$$

The components $\mathbf{o}_t$, $\mathbf{f}_t$, $\mathbf{i}_t$, $\mathbf{S}_t$ are standard RNNs and suffer from vanishing or exploding gradients. As for the recursive term $\frac{\partial \mathbf{C}_t}{\partial \theta}$, it is important to consider that $\mathbf{C}_t$ depends on $\mathbf{C}_{t-1}$ linearly before the nonlinearity is applied to it. If we assume vanishing gradients $\frac{\partial \mathbf{o}_t}{\partial \theta}, \frac{\partial \mathbf{f}_t}{\partial \theta}, \frac{\partial \mathbf{i}_t}{\partial \theta}, \frac{\partial \mathbf{S}_t}{\partial \theta}$, we get

$$\frac{\partial \mathbf{C}_t}{\partial \theta} \approx (\mathbf{f}_t \odot \mathbf{f}_{t-1} \odot \cdots \odot \mathbf{f}_{t-\tau}) \frac{\partial \mathbf{C}_{t-\tau-1}}{\partial \theta},$$

implying that error gradients between time steps are only subject to deletion by the forget gate. This effect is referred to as a Constant Error Carousel and enables LSTM networks to learn long-term dependencies [127].

Exploding gradients can be addressed as in standard RNNs, e.g. by using gradient clipping as described in Section 3.4.2.2.

3.6 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) architecture, introduced in 2014 [128], employs gating mechanisms similar to LSTM to tackle the vanishing gradient issue in standard RNNs. Distinct from LSTMs, GRUs feature only two gates, the *reset gate* and the *update gate*. Moreover, GRUs do not utilize a separate cell state $\mathbf{C}_t$, which means that information flows exclusively through the hidden state $\mathbf{h}_t$ in the unfolded network.

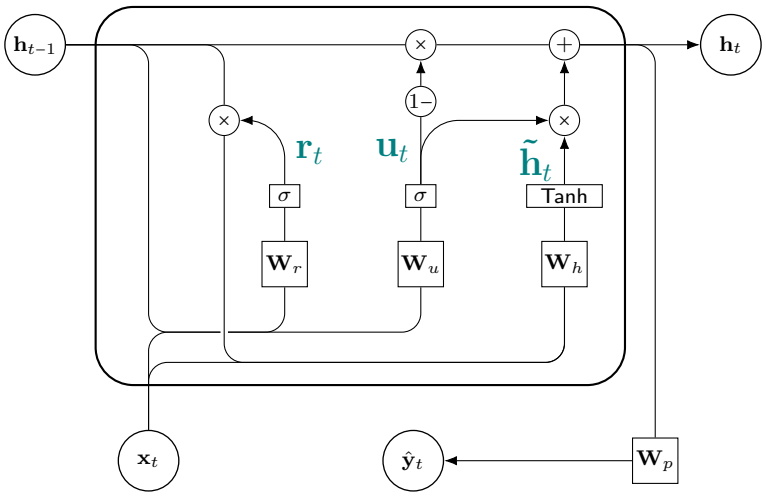


Figure 3.18: Structure of the GRU cell.

In time series forecasting applications, GRU networks can outperform LSTM networks in specific scenarios [129, 130, 131], and have also shown superior predictive performance regarding EV energy demand forecasting in [132], and comparable performance when predicting arrival and departure times at EV charging stations [56]. They require less time than LSTM networks to load and train due to a lower number of tuneable parameters. However, since GRUs are generally considered less expressive than LSTMs, the individual cost advantage of GRUs might be canceled out by the necessity for larger networks to achieve comparable expressiveness.

GRU networks are trained analogously to RNNs via forward pass, BPTT and parameter updates using a suitable optimization algorithm.

3.6.1 Forward Pass

Analogous to the previous sections, we consider the time-synchronous output setting. A GRU network is a nonlinear dynamical system that computes a vector of temporal features $\mathbf{h}_t \in \mathbb{R}^s$, called *hidden state*, at each time step t. The hidden state update is governed by

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \tilde{\mathbf{h}}_\mathbf{t},$$

where $\mathbf{u}_t$ and $\tilde{\mathbf{h}}_\mathbf{t}$ are called *update gate* and *candidate state*, respectively, and are computed via

$$\mathbf{u}_t = \sigma(\mathbf{W}_{hu} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{xu} \cdot \mathbf{x}_t + \mathbf{b}_u) \tag{3.53}$$

$$\tilde{\mathbf{h}}_t = tanh(\mathbf{W}_{hh} \cdot (\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{W}_{xh} \cdot \mathbf{x}_t + \mathbf{b}_h). \tag{3.54}$$

$\mathbf{r}_t$ is referred to as *reset gate* and given by

$$\mathbf{r}_t = \sigma(\mathbf{W}_{hr} \cdot \mathbf{h}_{t-1} + \mathbf{W}_{xr} \cdot \mathbf{x}_t + \mathbf{b}_r).$$

The prediction $\hat{\mathbf{y}}_t$ and the corresponding error $\mathcal{L}_t$ w.r.t. the input $\mathbf{x}_t$ and the target $\mathbf{y}_t$ are computed analogously to Equations 3.39 and 3.40.

In GRU networks, the hidden state $\mathbf{h}_t$ represents dynamic memory with read and update operations. The update gate can be interpreted as a bidirectional switch that determines the extent of information from past hidden states passed on to future hidden states. When $\mathbf{u}_t$ approaches $\mathbf{0}$, the current hidden state $\mathbf{h}_t$ effectively becomes a copy of $\mathbf{h}_{t-1}$. $\mathbf{u}_t$ is integrated with the reset gate, which governs how much infor-

mation from the previous hidden state should be discarded, through the candidate hidden state $\tilde{\mathbf{h}}_\mathbf{t}$. As $\mathbf{r}_t$ approaches $\mathbf{1}$, Equation 3.54 approximates Equation 3.38, corresponding to a standard RNN. For all entries of $\mathbf{r}_t$ that are close to 0, the candidate hidden state is the result of a fully connected layer with input $\mathbf{x}_t$. Any pre-existing hidden state is thus reset to defaults. Analogous to LSTM networks, the gates in GRU networks learn to regulate access to constant error flow during training.

3.6.2 Backpropagation Through Time

The derivation of the error gradient expressions required for updating parameters during gradient descent for training GRU networks follows the argumentation presented in Section 3.5.2 and involves similar steps.

At the final time step T, the error gradient with respect to the hidden state is given by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} = \mathbf{W}_p^\top \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_T} = \mathbf{W}_p^\top (\hat{\mathbf{y}}_T - \mathbf{y}_T).$$

For $0 \leq t < T$, every hidden state $\mathbf{h}_t$ has $\mathbf{h}_{t+1}$, $\mathbf{u}_{t+1}$, $\tilde{\mathbf{h}}_{t+1}$ and $\mathbf{r}_{t+1}$ as descendants and we have

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} &= \left(\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} + \left(\frac{\partial \mathbf{u}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{u}_{t+1}} + \left(\frac{\partial \tilde{\mathbf{h}}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_{t+1}} + \left(\frac{\partial \mathbf{r}_{t+1}}{\partial \mathbf{h}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{r}_{t+1}} \\
&= (\mathbf{1} - \mathbf{u}_t) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{hu}^\top \left(\mathbf{u}_{t+1} \odot (\mathbf{1} - \mathbf{u}_{t+1}) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{u}_{t+1}}\right) \\
&\quad + \mathbf{W}_{hh}^\top \left(\mathbf{r}_{t+1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_{t+1}^2) \odot \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_{t+1}}\right) + \mathbf{W}_{hr}^\top \left(\mathbf{r}_{t+1} \odot (\mathbf{1} - \mathbf{r}_{t+1}) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{r}_{t+1}}\right).
\end{aligned}$$

For $0 < t \leq T$, by examining the respective descendants of $\mathbf{u}_t$, $\tilde{\mathbf{h}}_t$ and $\mathbf{r}_t$, we derive

$$\frac{\partial \mathcal{L}}{\partial \mathbf{u}_t} = \left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{u}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = -\mathbf{h}_{t-1} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_t} = \left(\frac{\partial \mathbf{h}_t}{\partial \tilde{\mathbf{h}}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \mathbf{u}_t \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{r}_t} = \left(\frac{\partial \tilde{\mathbf{h}}_t}{\partial \mathbf{r}_t}\right)^\top \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_t} = \mathbf{W}_{hh}^\top \left(\mathbf{h}_{t-1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_t^2) \odot \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_t}\right).$$

The gradients with respect to the GRU cell's parameters can now be obtained by adding the contributions of all instantiations of a parameter $\theta^{(t)}$ in time, analogous to Section 3.4.2: The expressions $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_p}$ are computed just like in standard RNNs, as given in Equations 3.39 and 3.40. The derivation of gradients with respect to the remaining cell parameters can be found in Appendix B.

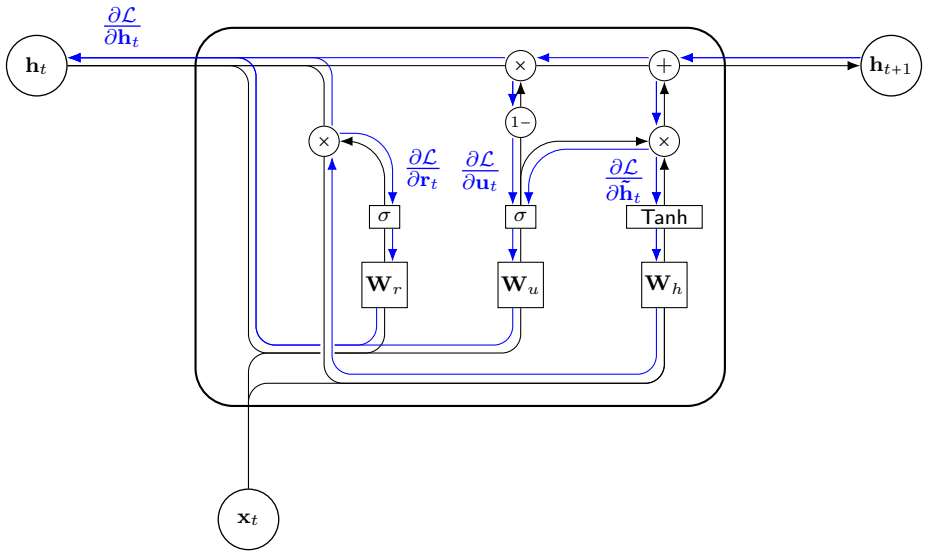


Figure 3.19: Gradient flow through a GRU cell.

Figure 3.19 illustrates the flow of gradients through a GRU cell. As in LSTM networks, the horizontal path between the hidden states located in the upper section of the cell equips GRU networks with long-term memory capabilities.

3.6.2.1 Constant Error Carousel

In a GRU network, $\mathbf{h}_t$ represents a dynamic memory unit whose state is controlled by the update gate:

$$\underbrace{\mathbf{memory}_t}_{\mathbf{h}_t} = \underbrace{\mathbf{forget}}_{(\mathbf{1}-\mathbf{u}_t)} \odot \underbrace{\mathbf{memory}_{t-1}}_{\mathbf{h}_{t-1}} + \underbrace{\mathbf{read}}_{\mathbf{u}_t} \odot \underbrace{\mathbf{features}_t}_{\tilde{\mathbf{h}}_t}.$$

Gradients of the hidden state in terms of a parameter θ that is involved in hidden state updates can be derived from Equation 3.53 using the product rule:

$$\frac{\partial \mathbf{h}_t}{\partial \theta} = \frac{\partial \mathbf{h}_{t-1}}{\partial \theta} - \frac{\partial \mathbf{u}_t}{\partial \theta} \odot \mathbf{h}_{t-1} - \mathbf{u}_t \odot \frac{\partial \mathbf{h}_{t-1}}{\partial \theta} + \frac{\partial \mathbf{u}_t}{\partial \theta} \odot \tilde{\mathbf{h}}_\mathbf{t} + \mathbf{u}_t \odot \frac{\partial \tilde{\mathbf{h}}_t}{\partial \theta}.$$

The components $\mathbf{u}_t$ and $\tilde{\mathbf{h}}_t$ are standard RNNs as discussed in Section 3.6.1 and can suffer from vanishing gradients. Assuming vanishing gradients $\frac{\partial \mathbf{u}_t}{\partial w_k}$, $\frac{\partial \tilde{\mathbf{h}}_t}{\partial w_k}$, we have

$$\frac{\partial \mathbf{h}_t}{\partial \theta} \approx \left((\mathbf{1} - \mathbf{u}_t) \odot (\mathbf{1} - \mathbf{u}_{t-1}) \odot \cdots \odot (\mathbf{1} - \mathbf{u}_{t-\tau})\right) \frac{\partial \mathbf{h}_{t-\tau-1}}{\partial \theta},$$

which implies that error gradients between time steps are only subject to deletion by the update gate.

3.7 Deep RNN Architectures

While depth in FFNs is defined as having multiple non-linear hidden layers, the concept is more ambiguous when considering RNNs, as any RNN can be expressed as a composition of several nonlinear layers when unfolded in time. The transitions hidden-to-hidden ($\mathbf{h}_{t-1} \rightarrow \mathbf{h}_t$), hidden-to-output ($\mathbf{h}_t \rightarrow \hat{\mathbf{y}}_t$) and input-to-hidden ($\mathbf{x}_t \rightarrow \mathbf{h}_t$) are considered shallow because they result from a linear transformation followed by an element-wise nonlinearity and contain no intermediate hidden nonlinear layer. The authors in [133] suggest defining deep RNNs by adding extra layers to these transitions, which results in

1. Deep input-to-hidden
2. Deep hidden-to-output
3. Deep hidden-to-hidden transitions.

A fourth way to construct a deep RNN architecture is stacking multiple recurrent hidden layers on top of each other [134], as depicted in Figure 3.20.

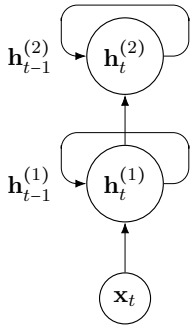


Figure 3.20: A simple deep recurrent neural network with two stacked hidden layers.

While being powerful in principle, recent research suggests that the deep transition approaches are more susceptible to gradient instabilities caused by long backpropagation paths and numerous authors consider stacked RNNs the go-to deep RNN architecture for time series forecasting tasks [101, 135, 136].

In a stacked RNN with L layers, the hidden states are denoted by $\mathbf{h}_t^{(l)}, l = 1, ...L$. Their updates are governed by

$$\mathbf{h}_t^{(l)} = \phi_l(\mathbf{W}_{hl} \cdot \mathbf{h}_{t-1}^{(l)} + \mathbf{W}_{xl} \cdot \mathbf{h}_t^{(l-1)} + \mathbf{b}_l),$$

where ϕ_l is the elementwise nonlinear activation function.

Stacking hidden layers on top of each other potentially allows the hidden state at each level to operate at a different timescale [137], resulting in more expressive models. Several recent studies suggest that stacked RNN architectures outperform their shallow counterparts on various time series forecasting tasks [138, 139], and energy demand forecasting tasks [140].

3.8 BiLSTM

A Bi-directional Long Short-Term Memory (BiLSTM) network is a type of RNN that processes input sequences in both directions [141]. It is made up of two LSTM layers, one that processes the input sequence in the forward direction and another that processes it in reverse.

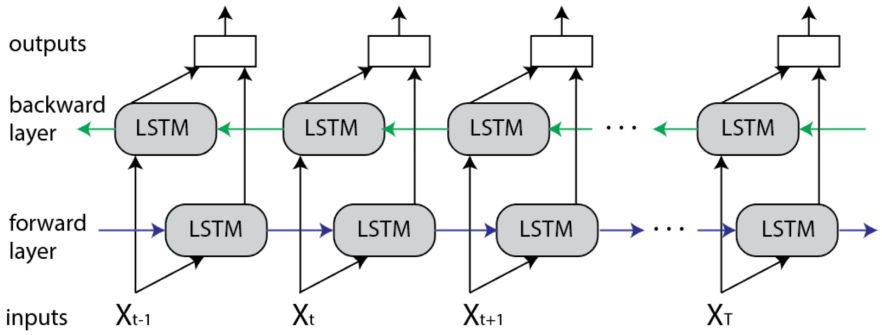


Figure 3.21: BiLSTM network [142].

In the time-synchronous output setting, at each time step t of the input sequence $\{\mathbf{x}_t\}_{t=1}^{T}$, the forward and backward LSTMs process the input in opposite directions to generate a sequence of hidden states, respectively denoted by

$$\overrightarrow{\mathbf{h}}_1, \overrightarrow{\mathbf{h}}_2, \ldots, \overrightarrow{\mathbf{h}}_T \quad \text{and} \quad \overleftarrow{\mathbf{h}}_1, \overleftarrow{\mathbf{h}}_2, \ldots, \overleftarrow{\mathbf{h}}_T.$$

The output at time t is then computed as the concatenation of the forward and backward hidden states, followed by a linear transformation and a nonlinear activation function ψ:

$$\hat{\mathbf{y}}_t = \psi\left(\mathbf{W}_p\left[\overrightarrow{\mathbf{h}}_t\,;\overleftarrow{\mathbf{h}}_t\right] + \mathbf{b}_p\right).$$

One of the main advantages of BiLSTM is that it can take into account the context from both the past and the future when making predictions, effectively increasing the amount of information available to the network.

Several recent studies come to the conclusion that BiLSTM outperforms regular LSTM in certain time series forecasting tasks [143, 144]. Notably, it was able to outperform a CNN-LSTM model in a residential energy demand forecasting task in [145].

3.9 CNN-LSTM

A CNN-LSTM network is a sequential integration of a CNN with an LSTM network, first proposed by Saianath et al. in [146]. The architecture leverages the complementary modeling capabilities of CNNs and LSTMs. The CNN component can disentangle variations within the input, which can then make it easier for the LSTM to learn temporal structure between successive time steps [133].

In its most simple setup, we assume no pooling or flatten layers and a causal convolutional layer with filters $f_i : \{0, ..., k-1\} \to \mathbb{R},\ i = 1, ..., n$. Given a univariate time series $\{x_t\}_{t=1}^{T}$ as input, at time step t, the causal convolutional layer outputs a set of feature maps

$$z_t^{(i)} = \sum_{j=0}^{k-1} f_i(j) \cdot x_{t-j}, \quad i = 1, ..., n.$$

Subsequently, the feature maps, denoted by $\mathbf{z}_t$, are fed into an LSTM layer, which operates as usual, i.e. it updates its hidden state $\mathbf{h}_t$ based on $\mathbf{z}_t$ and $\mathbf{h}_{t-1}$.

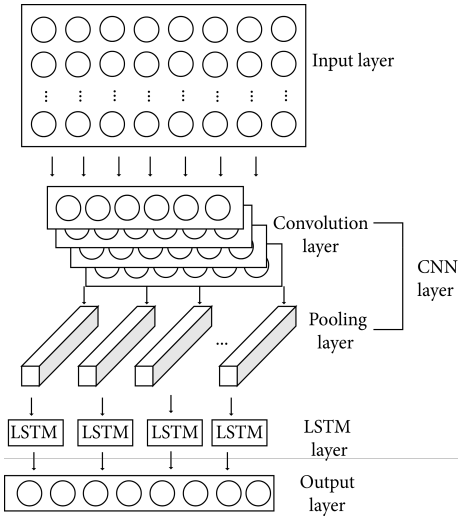


Figure 3.22: CNN-LSTM network [147].

In practice, CNN-LSTM is often combined with further fully connected layers, which help with mapping the features to a more separable space. The principal structure of a CNN-LSTM network is depicted in Figure 3.22.

The authors in several studies came to the conclusion that the sequential CNN-LSTM architecture outperforms simple CNN- and LSTM-architectures in certain time series forecasting tasks [147, 148, 149] and showed highly promising results in a residential energy demand forecasting task in [150], as well as in ultra-short term energy demand forecasting in [151].

3.10 Convolutional LSTM

Convolutional LSTM (ConvLSTM) was introduced by Shi et al. in [152]. In a ConvLSTM network, the matrix multiplication operations in the gate structures of the LSTM cell are replaced by convolutional operations, see Figure 3.23. This allows for processing of sequential data with spatial structure.

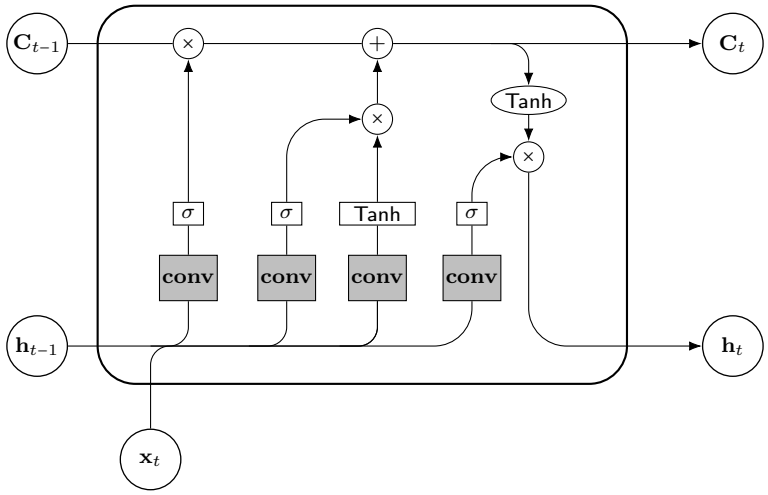


Figure 3.23: Structure of the ConvLSTM cell.

In its original formulation, the equations governing the hidden state updates in the ConvLSTM network are given by

$$
\begin{aligned}
\mathcal{F}_t &= \sigma(\mathbf{W}_{hf} * \mathcal{H}_{t-1} + \mathbf{W}_{xf} * \mathcal{X}_t + \mathcal{B}_f) \\
\mathcal{I}_t &= \sigma(\mathbf{W}_{hi} * \mathcal{H}_{t-1} + \mathbf{W}_{xi} * \mathcal{X}_t + \mathcal{B}_i) \\
\mathcal{O}_t &= \sigma(\mathbf{W}_{ho} * \mathcal{H}_{t-1} + \mathbf{W}_{xo} * \mathcal{X}_t + \mathbf{b}_o) \\
\mathcal{S}_t &= tanh(\mathbf{W}_{hc} * \mathcal{H}\mathcal{H}_{t-1} + \mathbf{W}_{xc} * \mathcal{X}_t + \mathbf{b}_c) \\
\mathcal{C}_t &= \mathcal{F}_t \odot \mathcal{C}_{t-1} + \mathcal{I}_t \odot \mathcal{S}_t \\
\mathcal{H}_t &= \mathcal{O}_t \odot tanh(\mathcal{C}_t),
\end{aligned}
$$

where "$*$" indicates the convolution operation, the weight matrices are 2D convolutional kernels and the gate outputs are 3D tensors [153].

Within this work, where a single observation $\mathbf{x}_t$ is a real-valued vector, the ConvLSTM architecture is implemented by replacing the LSTM gates' weight matrices with one-dimensional convolutional kernels.

The architecture showed promising results when applied to spatiotemporally correlated time series, which arise in domains such as weather forecasting, energy demand forecasting, and stock price prediction. In particular, the model outperformed a standard LSTM network in a precipitation forecasting task [152], and showed good results in residential energy demand forecasting [154].

3.11 Multi-Stream Deep Learning

The previously discussed CNN-LSTM and ConvLSTM architectures combine different layer types in a sequential manner to leverage their complementary modeling capabilities. An alternative approach to combining the strengths of layer types is *multi-stream deep learning*. This technique involves using multiple input streams, where each stream represents a different input modality or feature, and combining the outputs of these streams to make a final prediction.

In this thesis, we evaluate two different multi-stream deep learning models: the so-called *Hybrid LSTM* network and a parallel LSTM-CNN network or *PLCnet*.

3.11.0.1 Hybrid LSTM

The authors in [53] propose an architecture with two parallel input streams for predicting charging occupancy state sequences. The model takes into account both long-term tendencies of occupancy states and local temporal dependencies. Short-term charging state sequences are fed into an LSTM module while additional time-related features and long-term tendencies of charging state profiles are fed into a multilayer FFN. Their respective outputs are concatenated and fed into another fully connected layer and a dropout layer before the hybrid network generates a prediction. The architecture is depicted in Figure 3.24.

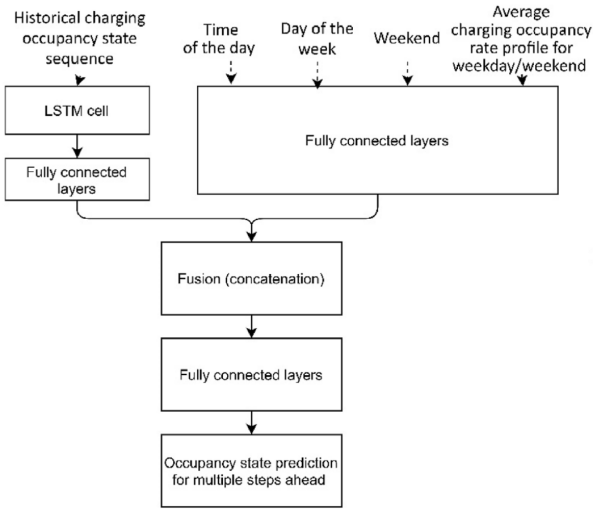


Figure 3.24: Hybrid LSTM architecture for occupancy state forecasting from [53].

According to the authors, Hybrid LSTM displayed state-of-the-art performance in charging station occupancy prediction at multiple forecasting horizons.

3.11.0.2 PLCnet

The authors in [48] propose a novel deep learning architecture called PLCnet (short for parallel LSTM-CNN network) to carry out short-term energy consumption prediction. PLCnet uses an LSTM network and a CNN in two parallel paths.

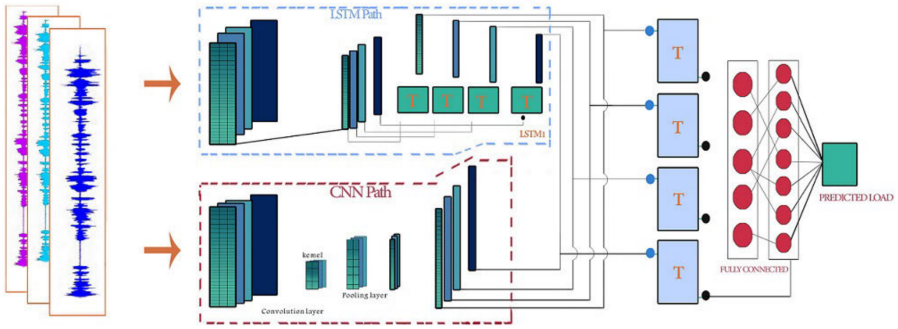


Figure 3.25: PLCnet architecture for short term load forecasting from [48].

In contrast to Hybrid LSTM, at each time step t, two copies of the same feature vector are fed into both paths simultaneously. After passing through the LSTM network and CNN, respectively, the outputs are concatenated and fed into another LSTM layer to learn temporal dependencies within outputs from both paths. A varying number of fully connected layers can be applied before the data is fed into a regularizing dropout layer, followed by a fully connected layer for making predictions.

The authors compared several deep learning architectures for load forecasting and found that the PLCnet architecture exhibited the strongest predictive performance across all assessed forecasting horizons. Notably, PLCnet achieved this performance while also demonstrating a comparatively short training time.

3.12 Windows and MIMO

As mentioned in Section 3.4, the deep learning-based models evaluated in this work are designed to operate on sequential data, which applies to both recurrent architectures and the TCN introduced in Section 3.3.5.1. They can be viewed as encoder-decoder architectures. Performing one-step-ahead forecasts using these models can be expressed as

$$\hat{\mathbf{y}}_t = g_{dec}(\mathbf{z}_t),$$

$$\mathbf{z}_t = g_{enc}(\mathbf{x}_{t-p}, ..., \mathbf{x}_t),$$

where p is referred to as the *look-back window*. Although RNNs theoretically do not require an explicit look-back window and can capture information from sequences of unlimited length, processing very long sequences with RNNs is computationally prohibitive for real-world applications. The look-back window for sequential models is similar to the window size p introduced in Section 3.2.1, where it is employed to convert a time series into a generic dataset for supervised learning. As mentioned in Section 3.4, the latent variable $\mathbf{z}_t$ corresponds to the hidden state $\mathbf{h}_t$ in RNNs. Meanwhile, for a TCN with a total of L causal convolutional layers, we have:

$$\mathbf{z}_t = \mathbf{a}_t^{(L)}, \tag{3.55}$$

where $\mathbf{a}_t^{(L)}$ denotes the activation of layer L at time step t, and it is easy to see that the look-back window corresponds to the receptive field of causal convolutional layer L.

In the previous discussions, we assumed time-synchronous output for simplicity. However, as time series forecasts using deep learning models are performed using the MIMO approach within this thesis, we want to briefly explore it. When forecasting h steps ahead, we get

$$(\hat{\mathbf{y}}_t, ..., , \hat{\mathbf{y}}_{t+h}) = g_{dec}(\mathbf{z}_t),$$

$$\mathbf{z}_t = g_{enc}(\mathbf{x}_{t-p}, ..., \mathbf{x}_t).$$

A common way to generate multistep outputs when learning from sequential data with deep learning models such as RNNs or TCNs is to simply incorporate a fully connected layer that outputs $(\hat{\mathbf{y}}_t, ..., \hat{\mathbf{y}}_{t+h})$, see Figure 3.26. To perform multistep forecasting, the model makes a prediction at time step t and then slides the window of fixed size p to the next position on the input data using a pre-defined stride and repeats

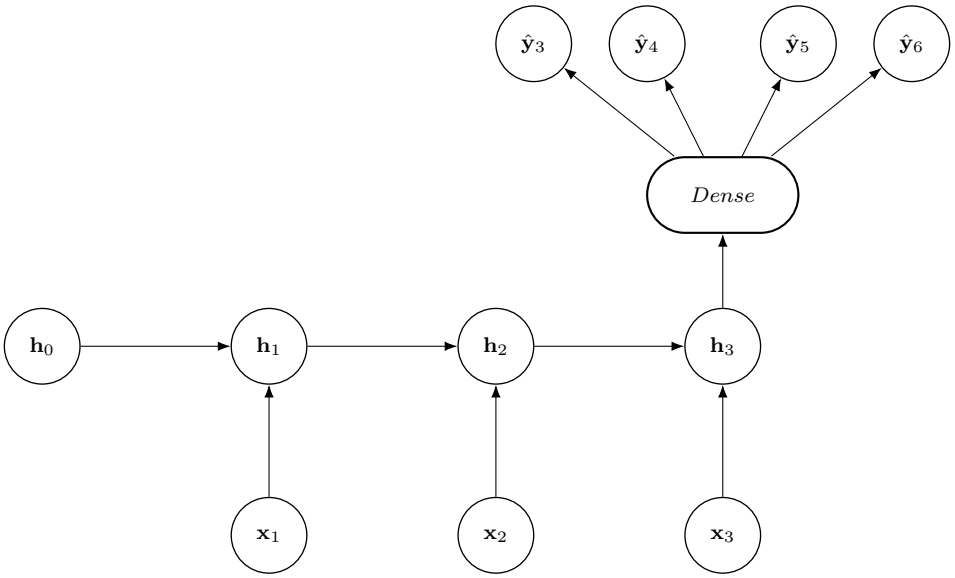


Figure 3.26: MIMO forecasting using a standard RNN.

the process. This *sliding window* approach allows for the generation of large amounts of training data, depending on the choice of stride. From a BPPT perspective, this means that for window size p and batch size 1 (for simplicity), in order to perform one parameter update, the network is unfolded across p time steps, and the error gradients are computed based on the prediction generated by processing p training examples.

This concludes the discussion of deep learning architectures that perform sequence modeling. In the next section, we discuss the chosen benchmark models, which operate within the classical supervised learning framework introduced in Section 3.1 and rely on datasets generated by the method described in Section 3.2.1.

3.13 Boosting

The decision to use AdaBoost and XGBoost as benchmark methods is primarily based on their ease of implementation compared to the deep neural architectures discussed previously, as well as their expected lower computational resource requirements during training [155]. XGBoost has demonstrated excellent performance in both regression and classification tasks involving EV-charging-related time series [10, 50], and has been shown to perform well on seasonal time series with the right features included in the feature set [156]. AdaBoost has shown promising results in time series classification [157], and can outperform XGBoost in certain energy demand forecasting tasks when appropriately modified [42]. The authors in [53] used it as a benchmark model for EV charging station occupancy forecasting.

Boosting was proposed in pioneering articles by Freund and Schapire during the 1990s [158, 159, 160] and remains one of the most powerful ideas introduced in machine learning. The original AdaBoost algorithm and its descendants have shown to be competitive in a variety of applications, are computationally fast and suitable for both classification and regression tasks. In 2001, Friedman developed a statistical framework that interprets boosting methods as numerical optimization in function space and generalizes them for arbitrary loss functions [161].

The basic idea of boosting is to combine many simple predictors to generate a powerful committee with improved performance over the single members. In general, boosting algorithms fit models from the class of adaptive basis function models, resulting in ensembles of the form

$$\begin{aligned} f(x) &= \sum_{m=0}^{M} f_m(x) \\ &= \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x), \end{aligned} \tag{3.56}$$

where $\phi_m \in \Phi,\ m = 1, ..., M$ are called *base hypotheses* or *base learners* and are selected from a suitable space of functions Φ. $\theta_0 = f_0(x)$ is some initial guess.

In the following, AdaBoost will be introduced by means of its original formulation, viewed as greedy optimization of a particular loss function. The connection to functional gradient descent will be made in the next section.

3.13.1 AdaBoost

AdaBoost stands for Adaptive Boosting and is considered the first practical boosting algorithm. It is based on an iterative additive procedure that constructs a probability distribution over the training data in every iteration, which is accomplished by adaptive re-weighting of training examples. Given training data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ from $\mathcal{X} \times \{-1, 1\}$, a number of iterations M, a zero-one loss function $\mathcal{I}$ and a class of basis functions Φ, AdaBoost outputs a classifier of the form

$$\hat{f}(x) = \hat{f}^{(M)}(x) = \sum_{m=1}^{M} \hat{\theta}_m \hat{\phi}_m(x),$$

where crisp classifications are given by $\hat{c}(x) = sgn(\hat{f}(x))$. The zero-one loss function is defined as

$$\mathcal{I}\left(f^{(m)}(x_i), y_i\right) = \mathbb{1}_{\{f^{(m)}(x_i) \neq y_i\}}.$$

After $m-1$ training steps, the classifier is given as a linear combination of weak classifiers of the form

$$\hat{f}^{(m-1)}(x) = \hat{\theta}_1 \hat{\phi}_1(x) + ... + \hat{\theta}_{m-1} \hat{\phi}_{m-1}(x).$$

At the m-th iteration, this classifier is extended by adding another weak classifier f_m:

$$\begin{aligned} f^{(m)}(x) &= \hat{f}^{(m-1)}(x) + f_m(x) \\ &= \hat{f}^{(m-1)}(x) + \theta_m \phi_m(x). \end{aligned}$$

If we let $w^{(m)}$ denote a yet to be constructed probability distribution over $\mathcal{D}$ that is used in the m-th iteration, we can define the $w^{(m)}$-weighted empirical risk of some $\hat{\phi}_m$ by

$$\epsilon_m := \sum_{i=1}^{N} w_i^{(m)} \mathbb{1}_{\{\hat{\phi}_m(x_i) \neq y_i\}}.$$

For now, we treat $\hat{\phi}_m$ as a black box and only assume that it performs at least as good as random guessing, i.e. $\epsilon_m \leq \frac{1}{2}$. We set

$$\hat{\theta}_m = \frac{1}{2} \ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right), \tag{3.57}$$

and observe that $\hat{\theta}_m \geq 0$ whenever $\epsilon_m \leq \frac{1}{2}$. The update rule for the probability distribution is now defined by

$$w_i^{(m+1)} = \frac{w_i^{(m)} e^{-y_i \hat{\theta}_m \hat{\phi}_m(x_i)}}{Z_m}, \tag{3.58}$$

where Z_m is a normalization factor such that $\sum_{i=1}^{N} w_i^{(m+1)} = 1$. This rule aims at increasing the weight for training examples (x_i, y_i) that were misclassified by $\hat{\phi}_m$. The particular choices of $\hat{\theta}_m$ and $w^{(m+1)}$ will be explained in the following.

Adaboost starts with a uniform distribution $w^{(1)}$ and computes $\epsilon_m, \hat{\theta}_m, \hat{\phi}_m$ and $w^{(m+1)}$ in each iteration $m = 1, ..., M$. The following theorem shows that the empirical risk can decrease rapidly with an increasing number of iterations M [57].

Theorem 4. *Let $\hat{f}$ be the hypothesis that AdaBoost returns after M iterations with associated intermediate weighted empirical risks $\epsilon_m \in [0,1]$, $m = 1,...,M$. Then the empirical risk of $\hat{f}$ on the training dataset $\mathcal{D}$ is bounded by*

$$\hat{R}(\hat{f}) \leq \prod_{m=1}^{M} 2\sqrt{\epsilon_m(1-\epsilon_m)}. \tag{3.59}$$

In particular, for $\gamma := \min\left\{\left|\frac{1}{2} - \epsilon_m\right|\right\}_{m=1}^{M}$, we have $\hat{R}(\hat{f}) \leq e^{-2\gamma^2 M}$.

Proof. We define $\hat{f}^{(M)} := \sum_{m=1}^{M} \hat{\theta}_m \hat{\phi}_m$. With $w_i^{(1)} = \frac{1}{N}$ we can write

$$w_i^{(M+1)} = w_i^{(1)} \cdot \frac{e^{-y_i\hat{\theta}_1\hat{\phi}_1(x_i)}}{Z_1} \cdots \frac{e^{-y_i\hat{\theta}_M\hat{\phi}_M(x_i)}}{Z_M} = \frac{e^{-y_i\hat{f}^{(M)}(x_i)}}{N\prod_{m=1}^{M} Z_m}.$$

We have

$$\hat{f}(x_i) \neq y_i \implies y_i\hat{f}^{(M)}(x_i) \leq 0 \implies e^{-y_i\hat{f}^{(M)}(x_i)} \geq 1,$$

and therefore

$$\hat{R}(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{\{\hat{f}(x_i)\neq y_i\}} \leq \frac{1}{N}\sum_{i=1}^{N} e^{-y_i\hat{f}^{(M)}(x_i)} = \prod_{m=1}^{M} Z_m,$$

where we used $\sum_{i=1}^{N} w_i^{(M+1)} = 1$. We can rewrite the normalization factors as

$$\begin{aligned} Z_m &= \sum_{i=1}^{N} w_i^{(m)} e^{-y_i\hat{\theta}_m\hat{\phi}_m(x_i)} = \sum_{\hat{\phi}_m(x_i)\neq y_i} w_i^{(m)} e^{\hat{\theta}_m} + \sum_{\hat{\phi}_m(x_i)=y_i} w_i^{(m)} e^{-\hat{\theta}_m} \\ &= \epsilon_m e^{\hat{\theta}_m} + (1-\epsilon_m)e^{-\hat{\theta}_m} = 2\sqrt{\epsilon_m(1-\epsilon_m)}, \end{aligned}$$

where we inserted $\hat{\theta}_m$ from Equation 3.57. This completes the proof of Equation 3.59. The second claim of the theorem is proved by using the fact that $1 - x \leq e^{-x}$ holds for all $x \in \mathbb{R}$. This allows us to bound

$$2\sqrt{\epsilon_m(1-\epsilon_m)} = \sqrt{1 - 4\left(\epsilon_m - \frac{1}{2}\right)^2} \leq \exp\left\{-2\left(\epsilon_m - \frac{1}{2}\right)^2\right\}.$$

□

Theorem 4 shows that if the error probabilities can be kept at least a constant γ away from $\frac{1}{2}$ (i.e. random guessing), the empirical risk will decrease exponentially fast in the number of iterations M [57].

When performing binary classification with AdaBoost, as illustrated in Figure 3.27, a popular approach is to restrict Φ to the set of decision stumps, which are trivial decision trees. Assuming $\mathcal{X} = \mathbb{R}^n$, a decision stump is defined as

$$\phi(\mathbf{x}) = \begin{cases} s & \text{if } x_k > c, \\ -s & \text{else.} \end{cases}$$

It has only three adjustable parameters, namely $c \in \mathbb{R}$, $k \in \{1, ..., n\}$ and $s \in \{-1, 1\}$. Decision stumps can be trained on a given training dataset using a simple brute-force approach: discretize the real numbers from the minimum to the maximum value in the training set, enumerate all possible classifiers and select the one with the lowest training error. More general tree models are introduced in Section 3.13.3.

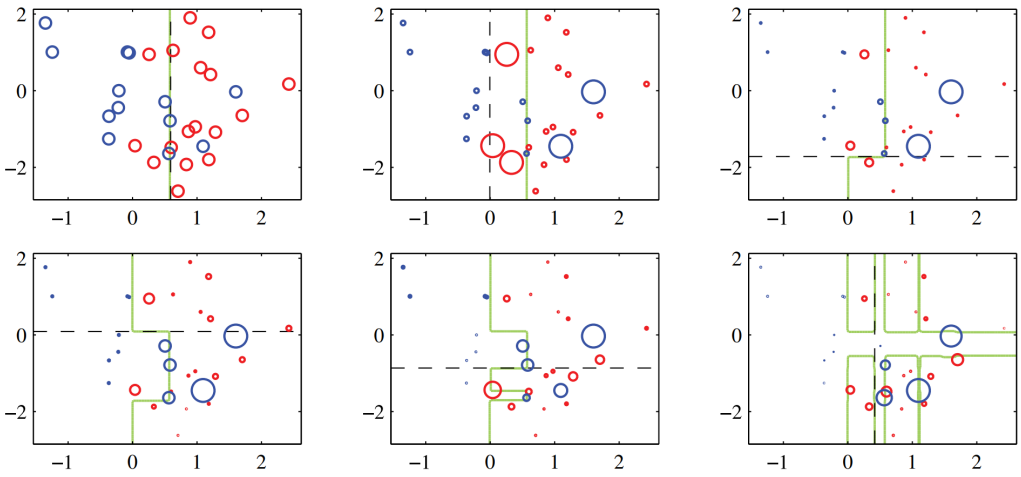


Figure 3.27: Illustration of AdaBoost. The decision boundary is depicted in green, the decision stump for each step is the dashed line. The results are shown after 1, 2, 3, 6, 10 and 150 steps [162].

The complete procedure for training a binary classifier using AdaBoost is outlined in Algorithm 3.

Algorithm 3: Discrete AdaBoost

Data: Training dataset $\mathcal{D}$, number of iterations M, set of weak learners Φ.

Result: Final classifier: $\hat{f}^{(M)}(x) = \sum_{m=1}^{M} \hat{f}_m(x)$.

for $i = 1, ..., N$ **do**
  Initialize weights:
    $w_i^{(1)} = \frac{1}{N}$
end
for $m = 1, ..., M$ **do**
  Fit a weak classifier:
    $\hat{\phi}_m(x) = \operatorname{argmin}_{\phi \in \Phi} \sum_{\phi(x_i) \neq y_i} w_i^{(m)}$
  Compute weighted empirical risk:
    $\epsilon_m = \sum_{i=1}^{N} w_i^{(m)} \mathbb{1}_{\{\hat{\phi}_m(\mathbf{x}_i) \neq y_i\}}$
  Compute step length:
    $\hat{\theta}_m = \frac{1}{2} \ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$
  Add to ensemble:
    $\hat{f}_m(x) = \hat{\theta}_m \hat{\phi}_m(x)$
  for $i = 1, ..., N$ **do**
    $w_i^{(m+1)} = \frac{w_i^{(m)} e^{-y_i \hat{\theta}_m \hat{\phi}_m(x_i)}}{Z_m}$
  end
end

AdaBoost can be modified in a straightforward way in order make it suitable for regression tasks. For example, AdaBoost.R2 uses regression trees as base learners and calculates an adjusted error $e_j^{(m)}$ at each step m:

$$D_m = \max_i \|y_i - \phi_m(x_i)\|_2,$$

$$e_j^{(m)} = \frac{\|y_j - \phi_m(x_j)\|_2}{D_m}, \quad j = 1, ..., N,$$

which is then used to update the distribution over $\mathcal{D}$. Details about the algorithm can be found in the original publication [163].

In general, boosting algorithms construct adaptive basis function models of the form given in Equation 3.56 by either approximately or exactly solving

$$\hat{f}_m = \underset{f_m \in \Omega_\Phi}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(\hat{f}^{(m-1)}(x_i) + f_m(x_i), y_i) \tag{3.60}$$

for a given loss function in each iteration m, where $\Omega_\Phi = span(\Phi)$ denotes the span of a set of base learners. This approach is referred to as *forward stagewise additive modeling* (FSAM) [164]. AdaBoost solves Equation 3.60 exactly for the exponential loss function under the constraint that Φ is the set of classifiers with $\mathcal{A} = \{-1, 1\}$.

The following section introduces a general framework to solve Equation 3.60 approximately for any suitable loss function by performing numerical optimization in function space.

3.13.2 Optimization in Function Space

The following derivations are based on [165]. In the first step, we introduce the population versions of the optimization algorithms used for boosting, where we rely on the true distribution from which given data was sampled. In the next step, the introduced methods will be applied to the practical case where we rely on the empirical risk $\hat{R}(f)$.

In the following, we assume that (X, Y) are random variables on $\mathbb{R}^n \times \mathbb{R}$ and the distribution of X and the conditional distribution of $Y|X$ are absolutely continuous w.r.t the Lebesgue measure.

3.13.2.1 Population Versions

The goal of boosting is to find a minimizer f^* of the risk $R(f)$, which is defined as

$$R(f) = \mathbb{E}_{X,Y}\left[\mathcal{L}(f(X), Y)\right],$$

where $\mathcal{L}(f, Y)$ is a loss function and f is a scalar-valued function in a Hilbert space $\mathcal{A}^{\mathcal{X}}$ with inner product $\langle \cdot, \cdot \rangle$, given by

$$\langle f, f \rangle = \mathbb{E}_X[f(X)^2].$$

The boosting approach assumes that f^* is within $\Omega_\Phi = span(\Phi)$, where Φ denotes a set of base learners $\phi_k : \mathbb{R}^n \to \mathbb{R}$:

$$f^* = \underset{f \in \Omega_\Phi}{\operatorname{argmin}} R(f).$$

The above objective is a convex minimization problem if the risk is convex in f, since Ω_Φ is also convex. Boosting finds f^* by FSAM, i.e.

$$f^{(m)}(x) = f^{(m-1)}(x) + \phi_m(x), \quad \phi_m \in \Phi, \quad m = 1, ..., M,$$

such that the risk is minimized:

$$\phi_m^* = \underset{\phi \in \Phi}{\operatorname{argmin}} R(f^{(m-1)} + \phi).$$

This minimization problem may not have an analytical solution, so an approximation method like gradient descent, Newton's method, or a hybrid variant is used to find an approximate solution.

For gradient boosting, we assume that the risk functional is Gâteaux differentiable for all $f \in \Omega_\Phi$. The Gâteaux derivative of R at f in the direction ϕ is defined as

$$\begin{aligned} dR(f;\phi) &= \frac{d}{d\tau} R(f + \tau\phi)\Big|_{\tau=0} \\ &= \lim_{\tau \to 0} \frac{R(f + \tau\phi) - R(f)}{\tau}, \quad f, \phi \in \Omega_\Phi. \end{aligned}$$

Gradient boosting chooses ϕ_m^* as the minimizer of the first-order Taylor approximation around $f^{(m-1)}$ with a norm-penalty imposed on the base learner:

$$\phi_m^* = \underset{\phi \in \Phi}{\operatorname{argmin}} R\left(f^{(m-1)}\right) + dR\left(f^{(m-1)};\phi\right) + \frac{1}{2}\langle\phi,\phi\rangle \tag{3.61}$$

$$= \underset{\phi \in \Phi}{\operatorname{argmin}} dR\left(f^{(m-1)};\phi\right) + \frac{1}{2}\langle\phi,\phi\rangle. \tag{3.62}$$

The penalty is added to include base functions ϕ of any norm. If we assume sufficient regularity[5] of $\mathcal{L}(f, Y)$, then $dR\left(f^{(m-1)};\phi\right)$ is given by

$$dR\left(f^{(m-1)};\phi\right) = \mathbb{E}_{X,Y}(g_m(X,Y)\phi(X)),$$

[5] $L(f, Y)$ is differentiable in f for P-almost all X and the derivative is integrable with respect to the measure of (X, Y).

where $g_m(X,Y)$ is the gradient of the loss function with respect to f at $f^{(m-1)}$:

$$g_m(X,Y) = \left.\frac{\partial \mathcal{L}(f,Y)}{\partial f}\right|_{f=f^{(m-1)}(X)}. \tag{3.63}$$

Consequently, we can write Equation 3.62 as

$$\phi_m^* = \operatorname*{argmin}_{\phi\in\Phi} \mathbb{E}_{X,Y}\left[g_m(X,Y)\phi(X) + \frac{1}{2}\phi(X)^2\right] \tag{3.64}$$

$$= \operatorname*{argmin}_{\phi\in\Phi} \mathbb{E}_{X,Y}\left[(-g_m(X,Y) - \phi(X))^2\right]. \tag{3.65}$$

This means that in gradient boosting, ϕ_m^* is the L_2-approximation of $-g_m(X,Y)$.

For Newton boosting, $R(f)$ is assumed to be two times Gâteaux differentiable. The second Gâteaux derivative of R in the direction ϕ is defined as

$$d^2R(f;\phi) = \left.\frac{d^2}{d\tau^2}R(f+\tau\phi)\right|_{\tau=0}, \quad f,\phi \in \Omega_\Phi.$$

Newton boosting chooses ϕ_m^* as the minimizer of the second-order Taylor approximation around $f^{(m-1)}$:

$$\phi_m^* = \operatorname*{argmin}_{\phi\in\Phi} R(f^{(m-1)}) + dR(f^{(m-1)};\phi) + \frac{1}{2}d^2R(f^{(m-1)};\phi). \tag{3.66}$$

If we assume sufficient regularity[6] of $\mathcal{L}(f,Y)$, then Equation 3.66 can be written as

$$\phi_m^* = \operatorname*{argmin}_{\phi\in\Phi} \mathbb{E}_{X,Y}\left[g_m(X,Y)\phi(X) + \frac{1}{2}h_m(X,Y)\phi(X)^2\right]$$

$$= \operatorname*{argmin}_{\phi\in\Phi} \mathbb{E}_{X,Y}\left[h_m(X,Y)\left(-\frac{g_m(X,Y)}{h_m(X,Y)} - \phi(X)\right)^2\right],$$

where $g_m(X,Y)$ is defined in Equation (3.63) and $h_m(X,Y)$ is the second derivative of the loss function with respect to f at $f^{(m-1)}$:

$$h_m(X,Y) = \left.\frac{\partial^2 \mathcal{L}(f,Y)}{\partial f^2}\right|_{f=f^{(m-1)}(X)}.$$

This means that in Newton boosting, ϕ_m^* is the weighted L_2-approximation of $-\frac{g_m(X,Y)}{h_m(X,Y)}$, where the weights are given by the Hessian $h_m(X,Y)$.

[6] $\mathcal{L}(f,Y)$ is twice differentiable in f for P-almost all X and the second derivative is integrable with respect to the measure of (X,Y).

3.13.2.2 Empirical Versions

We can obtain the empirical versions of gradient and Newton boosting by replacing the true risk functional with the empirical risk functional $\hat{R}(f)$, given by

$$\hat{R}(f^{(m)}) = \sum_{i=1}^{N} \mathcal{L}(\hat{f}^{(m-1)}(x_i) + \phi_m(x_i), y_i).$$

For gradient boosting, this results in the empirical version of Equation 3.65, which we can write as

$$\begin{aligned} \hat{\phi}_m &= \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} g_{m,i}\phi(x_i) + \frac{1}{2}\phi(x_i)^2 \\ &= \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} (-g_{m,i} - \phi(x_i))^2, \end{aligned}$$

where $g_{m,i}$ denotes the empirical gradient of the loss function with respect to f at $\hat{f}^{(m-1)}$ for observation i:

$$g_{m,i} = \frac{\partial}{\partial f}\mathcal{L}(f, y_i)\Big|_{f=\hat{f}^{(m-1)}(x_i)}.$$

Analogously, the empirical version of Equation (3.66) is given by

$$\begin{aligned} \hat{\phi}_m &= \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} g_{m,i}\phi(x_i) + h_{m,i}\frac{1}{2}\phi(x_i)^2 \\ &= \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} h_{m,i}\left(-\frac{g_{m,i}}{h_{m,i}} - \phi(x_i)\right)^2, \end{aligned}$$

where $h_{m,i}$ denotes the empirical Hessian of the loss function with respect to f at $\hat{f}^{(m-1)}$ for observation i:

$$h_{m,i} = \frac{\partial^2}{\partial f^2}\mathcal{L}(f, y_i)\Big|_{f=\hat{f}^{(m-1)}(x_i)}.$$

3.13.2.3 Line Search

The FSAM criterion (3.61) is often presented in the following form:

$$f^{(m)}(x) = f^{(m-1)}(x) + \rho_m \phi_m(x),$$

where, given a step direction $\hat{\phi}_m$, the step length $\rho_m \in \mathbb{R}$ is determined by a subsequent line search:

$$\rho_m = \underset{\rho \in \mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(\hat{f}^{(m-1)}(x_i) + \rho \hat{\phi}_m(x_i), y_i).$$

In the above derivations, line search was not considered explicitly since we assumed that Φ includes base learners of any norm. Furthermore, line search can oftentimes not be solved analytically and a second-order Taylor approximation is used instead. This implies that Newton boosting determines both step direction and step length at the same time.

3.13.2.4 AdaBoost as Gradient Boosting

The AdaBoost algorithm represents a special case of gradient boosting, where Φ is constrained to only include normalized base learners, and line search for finding the optimal step length can be solved analytically. At each iteration m, AdaBoost aims to find $\hat{\phi}_m$ such that

$$\hat{\phi}_m = \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} g_{m,i} \phi(x_i) + \frac{1}{2} \phi(x_i)^2. \tag{3.67}$$

The gradient of the exponential loss function for observation i is calculated as

$$g_{m,i} = -y_i e^{-y_i \hat{f}^{(m-1)}(x_i)}. \tag{3.68}$$

Plugging this expression into Equation 3.67 yields

$$\begin{aligned}
\hat{\phi}_m &= \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} \left(-y_i e^{-y_i \hat{f}^{(m-1)}(x_i)} + \frac{1}{2} \phi(x_i)^2 \right) \\
&= \underset{\phi \in \Phi}{\operatorname{argmin}} - \sum_{i=1}^{N} y_i \phi(x_i) e^{-y_i \hat{f}^{(m-1)}(x_i)} \\
&= \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{\phi_m(x_i) \neq y_i} w_i^{(m)} - \sum_{\phi_m(x_i) = y_i} w_i^{(m)} \\
&= \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{\phi_m(x_i) \neq y_i} w_i^{(m)},
\end{aligned}$$

where $w_i^{(m)}$ is defined in Equation 3.58 and we used $\phi(x_i)^2 = 1$ and $y_i\phi(x_i) \in \{-1, 1\}$, as well as

$$\sum_{\phi_m(x_i)=y_i} w_i^{(m)} = 1 - \sum_{\phi_m(x_i)\neq y_i} w_i^{(m)}.$$

The optimal stepsize ρ_n in AdaBoost can be found in closed form, as given in Equation 3.57.

The next section discusses a powerful machine learning algorithm that is based on Newton boosting.

3.13.3 XGBoost

XGBoost (eXtreme Gradient Boosting) and was introduced in 2016 [166]. It is a highly scalable tree-boosting algorithm and has rapidly become one of the best-performing and most used machine learning algorithms. Its key features are:

1. A unified approach for handling both classification and regression problems.
2. A regularization formalism for tree boosting that helps prevent overfitting.
3. The use of Newton's method for approximating the optimization problem.
4. Multiple algorithms for growing trees that can handle large datasets and sparse data, as well as parallelization capabilities [167].

XGBoost employs tree models as base learners, which are discussed in further detail in the subsequent section.

3.13.3.1 Tree Models

Tree models are simple, interpretable models that can be used for both classification and regression tasks. We will focus on the most frequently adapted choice of base learners, namely binary regression trees as introduced by Breiman et al. in [168]. These binary trees partition the feature space $\mathcal{X}$ into a set of T non-overlapping, rectangular regions $R_1, ..., R_T$ and fit a simple model in each region. Although it is possible to fit e.g. linear models in each region as described in [169], we will focus on the most common approach, where the simple models are constants. As the name suggests, tree models can be visualized as a tree structure, as shown in Figure 3.28. The node at the top of the tree is the *root node*, nodes with branches below them are called *splits* or *internal nodes* and the terminal nodes are called *leaves*. While a single tree model only has limited predictive power, combining multiple trees in *ensembles* yields very good predictive capabilities.

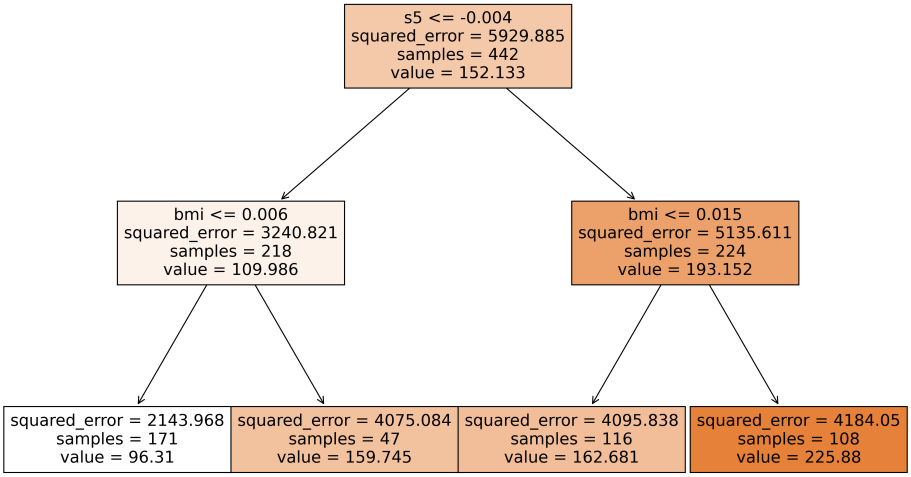


Figure 3.28: Visualization of a decision tree regressor trained on the `scikit-learn` diabetes dataset.

A binary regression tree on a training dataset $\mathcal{D}$ from $\mathcal{X} \times \mathcal{Y}$ is a mapping

$$f : \mathcal{X} \to \mathcal{Y}, \quad f(x) = \sum_{j=1}^{T} c_j \mathbb{1}_{\{x \in R_j\}},$$

and represents an adaptive basis function model with T basis functions or leaves, where $c_j \in \mathbb{R}$, $j = 1, ..., T$ are called leaf *weights*. The basis functions indicate whether observation x belongs in terminal node j. For $\mathcal{X} = \mathbb{R}^n$, the partition made by the

binary tree consists of rectangular regions of the form

$$R_j = (l_1, u_1] \times \cdots \times (l_n, u_n] \subset \mathbb{R}^n, \quad j = 1, ..., T,$$
$$-\infty \leq l_i < u_i \leq \infty, \quad i = 1, .., n.$$

The empirical risk of a single tree model can be expressed as

$$\hat{R}(f) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i), y_i) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\sum_{j=1}^{T} c_j \mathbb{1}_{\{x \in R_j\}}, y_i \right),$$

where $\mathcal{L}$ is an appropriately chosen loss function. In general, minimizing this objective is computationally infeasible. While obtaining the weights $c_i, ..., c_T$ for given regions $R_1, ..., R_T$ is typically easy, learning the optimal regions using this objective function is NP-complete [170]. The problem needs to be simplified by instead computing an approximate solution.

One computationally feasible approach to fit tree models is called Classification and Regression Trees (CART) and was introduced by Breiman et al. in [168]. CART uses a greedy recursive binary splitting procedure. It starts with a tree consisting of only one node and successively finds locally optimal branchings. Regression trees are typically fitted using squared error loss.

Given a region R_j, the weight $\hat{c}_j$ is estimated by

$$\hat{c}_j = \underset{c}{\operatorname{argmin}} \sum_{i \in I_j} \mathcal{L}(c, y_i), \quad \text{where } x_i \in R_j \text{ for } i \in I_j.$$

For squared error loss, $\hat{c}_j$ will be the mean of the responses in R_j. In the next step, CART learns the structure of the model $\hat{f}$ by minimizing

$$\hat{R}(\hat{f}) = \sum_{j=1}^{T} \sum_{i \in I_j} \mathcal{L}(\hat{c}_j, y_i) =: \sum_{j}^{T} \hat{L}_j,$$

where $\hat{L}_j$ is called *aggregated loss* at node j. If we let $\hat{f}_b$ denote the current tree model and $\hat{f}_a$ the model after a considered split at node k, resulting in a left node L and right node R, the empirical risk of these models is

$$\hat{R}(\hat{f}_b) = \sum_{j \neq k} \hat{L}_j + \hat{L}_k \quad \text{and} \quad \hat{R}(\hat{f}_a) = \sum_{j \neq k} \hat{L}_j + \hat{L}_L + \hat{L}_R.$$

By defining the *gain* of a considered split as

$$Gain = \hat{R}(\hat{f}_b) - \hat{R}(\hat{f}_a) = \hat{L}_k - \hat{L}_L - \hat{L}_R,$$

we can formulate a rule for choosing the best split: For each split made, compute the gain for every possible split at every possible node and choose the split with maximal gain [58].

Overfitting of CART can be avoided by adding a regularization term to the objective function or by pruning the final decision tree to an optimal depth. Details about these procedures can be found in [171].

Tree models have several advantages, including being easy to interpret, computationally inexpensive to construct, able to handle classification and regression tasks, robust to missing data and outliers, implicitly select features, capture nonlinear relationships, and scale well to large datasets. However, they also have several disadvantages, such as overfitting the training data, being unstable with high variance, lacking smoothness, and often having poor predictive performance [80, 164, 172]. A model class that addresses some of these issues is the class of additive tree models, which are discussed in the next section. These models use simple trees as base learners and inherit many of their advantages while reducing or eliminating some of their drawbacks.

3.13.3.2 Newton Tree-Boosting

In general, machine learning algorithms that use linear combinations of predictions from tree models as base learners are referred to as *additive tree models* or *tree ensembles*. For regression or binary classification tasks, they make predictions of the form

$$\hat{y} = \varphi\left(\theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x)\right),$$

where ϕ_m are regression trees and φ is a function depending on the learning problem at hand. For example, a common choice for binary classification is the logit function, while for regression problems, we have $\varphi(x) = e^x$ or $\varphi(x) = x$ [173].

Boosted tree models are adaptive basis function models with regression trees as basis

functions. Because trees are themselves adaptive basis function models, we get

$$\begin{aligned} f(x) &= \theta_0 + \sum_{m=1}^{M} \theta_m \sum_{j=1}^{T_m} \tilde{c}_{jm} \mathbb{1}_{\{x \in R_{jm}\}} \\ &= \theta_0 + \sum_{m=1}^{M} \sum_{j=1}^{T_m} c_{jm} \mathbb{1}_{\{x \in R_{jm}\}} \\ &= \theta_0 + \sum_{m=1}^{M} f_m(x). \end{aligned}$$

XGBoost employs stagewise additive modeling to fit an additive tree model. Given a dataset $\mathcal{D}$ from $\mathcal{X} \times \mathcal{Y}$, the algorithm initializes $\hat{f}_0(x) = \hat{\phi}_0(x) = \hat{\theta}_0$ to some constant value, i.e.

$$\hat{\theta}_0 = \underset{\theta \in \mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(\theta, y_i),$$

where $\mathcal{L}$ is a differentiable convex loss function chosen for the learning problem at hand. At each iteration $m \in \{1, ..., M\}$, the algorithm aims to estimate a base learner $\hat{\phi}_m$ such that

$$\hat{\phi}_m = \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(\hat{f}^{(m-1)}(x_i) + \phi(x_i), y_i) + \Omega(\phi),$$

where Φ is the set of functions of the form

$$\phi(x) = \sum_{j=1}^{T} c_j \mathbb{1}_{\{x_i \in R_j\}},$$

and $\Omega(\phi)$ is a regularization term defined as

$$\Omega(\phi) = \gamma T + \frac{\lambda}{2} \sum_{j=1}^{T} c_j^2.$$

It is easy to see that Ω penalizes the number of leaves of each individual tree through γ and imposes L_2-regularization on leaf weights.

XGBoost employs Newton boosting to fit its base estimators, which means we can rewrite the FSAM criterion stated in Equation 3.69 as

$$\hat{\phi}_m = \underset{\phi \in \Phi}{\operatorname{argmin}} \sum_{i=1}^{N} \left(g_{m,i} \phi(x_i) + \frac{1}{2} h_{m,i} \phi(x_i)^2 \right) + \gamma T + \frac{\lambda}{2} \sum_{j=1}^{T} c_{jm}^2.$$

By expressing the objective in terms of trees as basis functions and using the disjoint nature of the regions of the tree leaves, we get

$$\sum_{i=1}^{N}\left(g_{m,i}\sum_{j=1}^{T}c_{jm}\mathbb{1}_{\{x_i\in R_{jm}\}}+\frac{1}{2}h_{m,i}\left(\sum_{j=1}^{T}c_{jm}\mathbb{1}_{\{x\in R_{jm}\}}\right)^2\right)+\gamma T+\frac{\lambda}{2}\sum_{j=1}^{T}c_{jm}^2 \tag{3.69}$$

$$=\sum_{j=1}^{T}\left(\sum_{i\in I_{jm}}g_{m,i}c_{jm}+\frac{1}{2}\sum_{i\in I_{jm}}h_{m,i}c_{jm}^2\right)+\gamma T+\frac{\lambda}{2}\sum_{j=1}^{T}c_{jm}^2 \tag{3.70}$$

$$=\sum_{j=1}^{T}(G_{jm}c_{jm}+\frac{1}{2}(H_{jm}+\lambda)c_{jm}^2)+\gamma T, \tag{3.71}$$

where

$$G_{jm}=\sum_{i\in I_{jm}}g_{m,i}\quad\text{and}\quad H_{jm}=\sum_{i\in I_{jm}}h_{m,i}.$$

For a proposed, fixed structure, the weights that minimize Expression 3.71 are given by

$$\hat{c}_{jm}=-\frac{G_{jm}}{H_{jm}+\lambda},\quad j=1,..,T. \tag{3.72}$$

We can now learn the structure of the tree following the procedure described in the previous section. To this means, we plug in the weights from Equation 3.72 into Expression 3.71 and find the criterion for determining the structure to be

$$\{\hat{R}_{jm}\}_{j=1}^{T}=\underset{\{R_{jm}\}_{j=1}^{T}}{\operatorname{argmin}}\sum_{j=1}^{T}(-\frac{1}{2}\frac{G_{jm}^2}{H_{jm}+\lambda})+\gamma T.$$

The optimal split can now be found by maximizing the gain, which is given by

$$Gain=\frac{1}{2}\left(\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{G_{jm}^2}{H_{jm}+\lambda}\right)-\gamma.$$

The final weights for the learned structure are already found when searching for the structure and are given in Equation 3.72 [58].
When fitting the base learner is complete, $\hat{\phi}_m$ is added to the ensemble via

$$\hat{f}_m(x)=\eta\hat{\phi}_m(x),$$
$$\hat{f}^{(m)}(x)=\hat{f}^{(m-1)}(x)+\hat{f}_m(x),$$

where $\eta\in(0,1]$ is called the *learning rate*.

The full regularized Newton tree-boosting procedure used by XGBoost is outlined in Algorithm 4.

Algorithm 4: Newton Tree-Boosting

Data: Training dataset $\mathcal{D}$, loss function $\mathcal{L}$, number of iterations M, learning rate η

Result: Final model $\hat{f}(x) = \hat{f}^M(x) = \sum_{m=0}^{M} \hat{f}_m(x)$

Initialize model with a constant value:

$\hat{f}^0(x) = \hat{f}_0(x) = \hat{\theta}_0 = \text{argmin}_{\theta \in \mathbb{R}} \sum_{i=1}^{N} \mathcal{L}(\theta, y_i)$

for $m = 1, ..., M$ **do**

  for $i = 1, ..., N$ **do**

    Compute empirical gradients:

    $g_{m,i} = \frac{\partial}{\partial f} \mathcal{L}(f, y_i)\Big|_{f=\hat{f}^{(m-1)}(x_i)}$

    Compute empirical Hessians:

    $h_{m,i} = \frac{\partial^2}{\partial f^2} \mathcal{L}(f, y_i)\Big|_{f=\hat{f}^{(m-1)}(x_i)}$

  end

  Fit a new decision tree by minimizing the loss function:

  Find optimal tree structure $\{\hat{R}_{jm}\}_{j=1}^{T}$ by finding splits which maximize

  $Gain = \frac{1}{2}\left(\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G_{jm}^2}{H_{jm}+\lambda}\right) - \gamma,$

  where $G_{jm} = \sum_{i \in I_{jm}} g_{m,i}$ and $H_{jm} = \sum_{i \in I_{jm}} h_{m,i}$

  Determine leaf weights $\{\hat{c}_{jm}\}_{j=1}^{T}$ for learned structure:

  $\hat{c}_{jm} = -\frac{G_{jm}}{H_{jm}+\lambda}$

  New tree: $\hat{\phi}_m(x_i) = \sum_{j=1}^{T} \hat{c}_{jm} \cdot \mathbb{1}_{\{x_i \in \hat{R}_{jm}\}}$

  Update model: $\hat{f}_m(x) = \eta \hat{\phi}_m(x)$

end

Boosting algorithms based on tree models can be computationally expensive due to the cost of growing the regression trees, as noted in [174]. XGBoost aims to reduce computational complexity by avoiding the need for repeated data sorting at each node. Instead, XGBoost employs a compressed column-based data structure that allows the data to be pre-sorted, reducing the need for sorting each attribute repeatedly. This structure also enables the algorithm to find the best split for each attribute in parallel. Additionally, XGBoost uses a method based on percentiles of the dataset to reduce the number of candidate splits that have to be evaluated. This approach tests only a subset of candidate splits and computes their gain using aggregated statistics, rather than scanning through all possible splits [58].

3.14 Baseline Models

The use of basic machine learning models such as Linear Regression for regression problems and Logistic Regression for classification problems is common in evaluating the performance of deep learning-based forecasters [10, 48, 53, 150, 175]. These models are known for their ease of implementation and low computational requirements, making them a popular choice as baseline models.

3.14.1 Linear Regression

As mentioned in section 3.1, a popular model class for solving supervised learning tasks is the class of linear models

$$\mathcal{F} = \{f : f(x) = \theta_0 + \sum_{k=1}^{n} \theta_k x_k, \quad \forall x \in \mathcal{X}\}.$$

The approach of using a linear model to solve a regression task is referred to as *Linear Regression*. The parameter estimation task simplifies to finding a vector $\hat{\boldsymbol{\theta}} = (\hat{\theta}_0, \hat{\theta}_1, ..., \hat{\theta}_n) \in \mathbb{R}^{n+1}$. If squared error loss is used for estimating $\hat{\boldsymbol{\theta}}$ from a training dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$ from $\mathbb{R}^{n \times m}$, i.e.

$$\mathcal{L}(f(\mathbf{x}_i), \mathbf{y}_i) = \frac{1}{2} \| f(\mathbf{x}_i) - \mathbf{y}_i \|^2,$$

where $f : \mathbb{R}^n \to \mathbb{R}^m$, the parameter estimation method is referred to *ordinary least squares* (OLS). In OLS, the parameters can be determined analytically by solving

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y}, \tag{3.73}$$

where $\mathbf{X} = (\mathbf{x}_1, ..., \mathbf{x}_N)^\top$ is called *design matrix* and $\mathbf{Y} = (\mathbf{y}_1, ..., \mathbf{y}_N)^\top$. Equation 3.73 has a unique solution if $\mathbf{X}^\top \mathbf{X}$ is invertible. In the singular case, the parameters $\hat{\boldsymbol{\theta}}$ are not uniquely determined and can be estimated using a pseudo-inverse. The estimated parameters can be used to make predictions as follows:

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\theta}} = \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}\mathbf{y}.$$

In practice, the parameters are usually estimated using an iterative optimization algorithm. SGD is a popular choice for high-dimensional Linear Regression problems

[164]. In order to formulate a parameter update rule, we have to derive $\nabla\mathcal{L}_i(\boldsymbol{\theta})$, where

$$\mathcal{L}_i : \mathbb{R}^{n+1} \to \mathbb{R}, \quad \boldsymbol{\theta} \mapsto \mathcal{L}\left(f(\mathbf{x}_i;\boldsymbol{\theta}), \mathbf{y}_i\right).$$

By adding the *intercept terms* $x_{i,0} = 1,\ i = 1, ..., N$, we can derive

$$\nabla\mathcal{L}_i = \left(\frac{\partial\mathcal{L}_i}{\partial\theta_0} \cdots \frac{\partial\mathcal{L}_i}{\partial\theta_n}\right)^\top$$

by using the chain rule:

$$\begin{aligned}
\frac{\partial\mathcal{L}_i}{\partial\theta_l} &= \frac{\partial}{\partial\theta_l}\frac{1}{2}\|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2 \\
&= (\hat{\mathbf{y}}_i - \mathbf{y}_i)\frac{\partial}{\partial\theta_l}(\hat{\mathbf{y}}_i - \mathbf{y}_i) \\
&= (\hat{\mathbf{y}}_i - \mathbf{y}_i)\frac{\partial}{\partial\theta_l}\left(\sum_{k=1}^{n}\theta_k x_{i,k} - \mathbf{y}_i\right) \\
&= (\hat{\mathbf{y}}_i - \mathbf{y}_i)x_{i,l} \quad l = 0, ..., n.
\end{aligned}$$

$\hat{\boldsymbol{\theta}}$ can now be estimated by mini-batch SGD, using the gradient

$$\mathbf{g}_m^{(i:i+b)} = \frac{1}{b}\sum_{i\epsilon B_m}\nabla\mathcal{L}_i\left(\boldsymbol{\theta}^{(m-1)}\right).$$

In the statistics literature, it is often assumed that the linear model is the true underlying data-generating process, i.e. the examples are assumed to be sampled from the data-generating process

$$\mathbf{y} = \theta_0 + \sum_{k=1}^{n}\theta_k x_k + \epsilon,$$

where ϵ denotes a noise term. Under certain assumptions and when the correct model is chosen, OLS regression has several desirable properties, including unbiasedness and consistency [176].

3.14.2 Logistic Regression

Logistic Regression is a statistical model from the class of generalized linear models [177] that can be used as a supervised learning algorithm for classification. In order to derive how Logistic Regression estimates parameters from training data, we examine the relationship between ERM and maximum likelihood estimation (MLE): Given observations $y_1, ..., y_N$ of a random variable Y with the underlying parametrized distribution $Y \sim P_Y(y; \boldsymbol{\theta})$, the parameters $\boldsymbol{\theta} \in \Theta$ can be estimated using MLE [58]:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}}\, l(\boldsymbol{\theta}; y_i, ..., y_N) = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}} \sum_{i=1}^{N} \log P_Y(y_i; \boldsymbol{\theta}),$$

where the parameter space Θ is a finite-dimensional subset of Euclidean space. If we let $\boldsymbol{\theta}$ be a function of X, i.e. $\boldsymbol{\theta} : \mathcal{X} \to \Theta$, and assume

$$Y|X \sim P_{Y|X}(y; \boldsymbol{\theta}(X)),$$

then it follows that

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\boldsymbol{\theta} \in \Theta^{\mathcal{X}}}{\operatorname{argmax}} \sum_{i=1}^{N} \log P_{Y|X}(y_i; \boldsymbol{\theta}(\mathbf{x}_i)) \\ &= \underset{\boldsymbol{\theta} \in \Theta^{\mathcal{X}}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} -\log P_{Y|X}(y_i; \boldsymbol{\theta}(\mathbf{x}_i)).\end{aligned}$$

This means that $\hat{\boldsymbol{\theta}}$ minimizes the empirical risk of the likelihood-based loss function

$$\mathcal{L}(y, \boldsymbol{\theta}(\mathbf{x})) = -\log P_{Y|X}(y; \boldsymbol{\theta}(\mathbf{x})).$$

For a bounded parameter space Θ, generalized linear models are typically fit on a transformed scale, which makes estimation more convenient. This is accomplished by defining a *link function* $g : \Theta \to \mathcal{A}$ and then estimating the function

$$f(\mathbf{x}) = g(\boldsymbol{\theta}(\mathbf{x})) \in \mathcal{A}.$$

By applying g^{-1} to the predictions, the final model can be used for making predictions on the original scale.

For binary classification, i.e. $\mathcal{Y} = \{0, 1\}$, we can assume $Y|X \sim Bernoulli(p(X)$ with the corresponding likelihood-based loss function

$$\mathcal{L}(y, p(\mathbf{x})) = -y \log(p(\mathbf{x})) - (1 - y) \log(1 - p(\mathbf{x})). \tag{3.74}$$

It is easy to show that the function which minimizes the risk is

$$p^*(\mathbf{x}) = \Pr(Y = 1|X = \mathbf{x}).$$

It is common to model the bounded function $p(\mathbf{x})$ on the logit scale by transforming it using the *logit link*, which is the inverse of the standard logistic function:

$$f(\mathbf{x}) = \sigma^{-1}(p(\mathbf{x})) = \log \frac{p(\mathbf{x})}{1 - p(\mathbf{x})}.$$

The loss function for the model on the transformed scale is

$$\mathcal{L}(f(x), y) = \log\left(1 + e^{f(x)}\right) - yf(x).$$

As a (generalized) linear model, Logistic Regression assumes that the conditional probability of observing a positive label y_i, given a particular training example $\mathbf{x}_i$, has the parametric form

$$\Pr(y_i = 1|\mathbf{x}_i; \boldsymbol{\theta}) = \sigma\left(\theta_0 + \sum_{k=1}^{n} \theta_k x_k\right) = \frac{1}{1 + e^{-\boldsymbol{\theta}^\top \mathbf{x}}}.$$

By making use of the likelihood-based loss function from Equation 3.74, the empirical risk of the model can be written as

$$R(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} (y_i \log(\sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)) + (1 - y_i) \log(1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i))).$$

The optimal parameter configuration $\hat{\boldsymbol{\theta}}$ can be estimated by maximizing the likelihood, which is equivalent to minimizing the negative log-likelihood, by applying a suitable iterative optimization algorithm. A traditional choice here is Iteratively Reweighted Least Squares (IRLS), which is based on Newton's method. However, especially for large datasets, using mini-batch SGD instead of IRLS can significantly improve computation time [178]. The model's loss for one training example is

$$\mathcal{L}_i(\boldsymbol{\theta}) = -y_i \log(\sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)) - (1 - y_i) \log(1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)).$$

The entries of $\nabla \mathcal{L}_i(\boldsymbol{\theta})$ can be derived analogously to section 3.14.1 by using the chain rule:

$$\begin{aligned}
-\frac{\partial \mathcal{L}_i(\boldsymbol{\theta})}{\partial \theta_j} &= \left(y_i \frac{1}{\sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)} - (1 - y_i) \frac{1}{1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)} \right) \frac{\partial}{\partial \theta_j} \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i) \\
&= \left(y_i \frac{1}{\sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)} - (1 - y_i) \frac{1}{1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)} \right) \sigma\left(\boldsymbol{\theta}^\top \mathbf{x}_i\right) \left(1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)\right) \frac{\partial}{\partial \theta_j} \boldsymbol{\theta}^\top \mathbf{x}_i \\
&= \left(y_i (1 - \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)) - (1 - y_i) \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)\right) x_j^{(i)} \\
&= \left(y_i - \sigma(\boldsymbol{\theta}^\top \mathbf{x}_i)\right) x_j^{(i)}, \quad j = 0, 1, ..., n.
\end{aligned}$$

The log-likelihood can now be iteratively maximized via mini-batch SGD using the parameter update rule

$$\boldsymbol{\theta}^{(m)} = \boldsymbol{\theta}^{(m-1)} - \gamma_m \mathbf{g}_m^{(i:i+b)},$$

$$\mathbf{g}_m^{(i:i+b)} = \frac{1}{b} \sum_{i \in B_m} \nabla \mathcal{L}_i \left(\boldsymbol{\theta}^{(m-1)} \right),$$

analogous to Equation 3.22.

3.15 Evaluation Metrics

A range of metrics can be applied to evaluate the performance of forecasting models. In the following, we explore the most commonly used metrics for each specific task.

3.15.1 Regression

For regression tasks, the most frequently used error metrics are mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE) and mean absolute percentage error (MAPE) [15, 179]:

$$MAE = \frac{\sum_{i=1}^{N} |y_i - x_i|}{N},$$

$$MAPE = \frac{100}{N} \sum_{i=1}^{N} \left| \frac{y_i - x_i}{x_i} \right|,$$

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - x_i)^2,$$

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - x_i)^2},$$

where y_i is the predicted value and x_i is the actual value and N is the number of test examples. MAE is the easiest to understand and compute. However, it cannot be compared across series because it is scale-dependent. Percentage errors have the advantage of being scale independent, so they are frequently used to compare forecast performance between different data sets. However, measurements based on percentage errors have the disadvantage of being infinite or undefined if zero values exist in a series. MSE and RMSE provide a quadratic loss function and are useful when the consequences of large errors are proportionately more severe than those of equivalent smaller ones. However, they are absolute measures influenced strongly by extreme values and are scale-dependent, which makes using them for comparisons across datasets, forecasting horizons and methods highly problematic [180].

A metric that addresses the issues above was proposed in [181] and is denoted mean absolute scaled error (MASE). It is computed as the MAE of the forecast values divided by the MAE of a one-step naïve forecast. The naïve forecast is generated at any

step by equating the current forecast to the output from the last time step:

$$MAE_{naïve} = \frac{1}{N-1}\sum_{i=2}^{N}|y_i - y_{i-1}|$$

$$MASE = \frac{MAE}{MAE_{naïve}}.$$

MASE is scale-invariant and penalizes small and large and negative and positive errors equally. It is easy to interpret: values greater than one indicate that one-step forecasts from the naïve method perform better than the examined model's forecasts. When comparing forecasting methods, a lower MASE value is preferred. MASE can be adjusted for time series with a seasonal component by calculating the $MAE_{naïve}$ for a seasonally adjusted naïve method. According to [181], MASE can be used to compare models on a single time series and forecast accuracy between time series. It is the accuracy metric of choice within this thesis. RMSE is employed as an auxiliary metric for the regression task.

3.15.2 Classification

Prediction of charging station occupancy states is a discrete binary time series forecasting problem. Model performance can be evaluated via standard evaluation metrics for classification models [10, 53]. Several metrics can be derived from the *confusion matrix*, a table that displays the number of correctly classified and misclassified examples, as shown in Figure 3.29.

The most commonly used metric for classifiers is accuracy, defined by

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}.$$

Accuracy computes the proportion of correctly classified cases among all cases. It is inappropriate for models trained on imbalanced datasets. In these scenarios, the F1-score can be used as an additional metric to evaluate model performance. The F1-score takes into account both recall and precision, which are discussed in the following.

Recall is also called sensitivity or true positive rate and is given by

$$Recall = \frac{TP}{TP + FN}.$$

It is the proportion of positive cases that were correctly classified and is useful when the cost of false negative predictions is high.

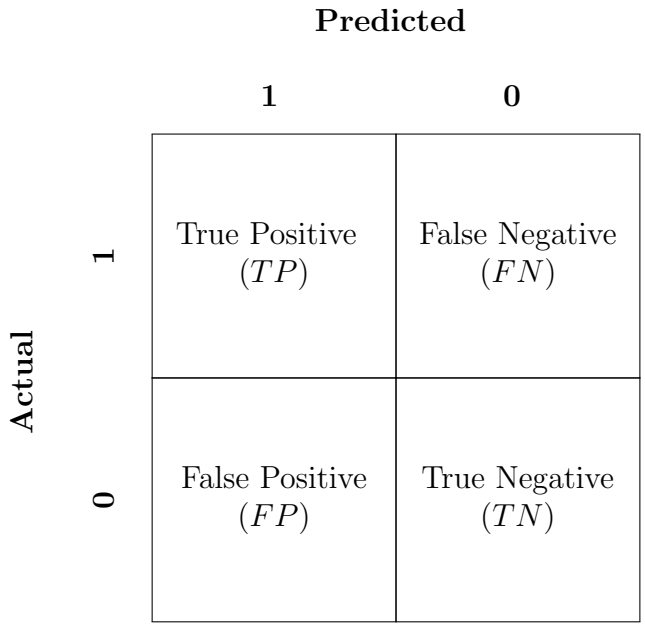


Figure 3.29: Confusion matrix of a binary classification model.

Precision is the proportion of predicted positive samples that are truly positive:

$$Precision = \frac{TP}{TP + FP}.$$

It is also called positive predictive value and is useful when the cost of false positive predictions is high.

A model with high recall and low precision classifies most of the positive samples correctly but has many false positives. On the other hand, a model with high precision and low recall is accurate when classifying a sample as positive but can only classify a few positive samples. The F1-score is the harmonic mean of recall and precision, given by

$$F1 = 2 \cdot \frac{Recall \cdot Precision}{Recall + Precision}.$$

A high F1-score indicates that both precision and recall are high and provides a single measure of the model's performance on both classes [182].

To provide a thorough evaluation of the binary classification models under consideration, we employ a combination of accuracy and F1-score for classification assessment within this work.

Chapter 4

Methods

This thesis considers two approaches to time series forecasting that differ depending on the addressed scenario. Specifically, energy demand prediction constitutes a regression problem, referred to as *multi-response regression* in the case of multiple real-valued output variables. In contrast, occupancy prediction involves modeling a sequence of binary occupancy states for an individual charging station, and hence it is treated as a binary classification problem. In the case of multiple binary output variables, the task is referred to as *multi-label classification* [183].

We decide to add exogenous variables as additional features to the original univariate time series, for several reasons. Firstly, recent studies have demonstrated that EV energy demand and charging station occupancy are strongly influenced by the behavior of EV users, which is in turn influenced by various factors such as weather, temperature, time of day, day type, and public holidays [10, 51, 53, 120, 184]. Secondly, machine learning models are data-driven and profit from large amounts of training data. They are well-suited for identifying complex patterns in large, high-dimensional feature sets, and are capable of detecting hidden relationships between the input variables that are difficult or impossible to identify using traditional methods. The inclusion of exogenous variables has been shown to improve predictive performance in numerous time series forecasting tasks [185, 186].

Also, we account for non-stationary patterns in the data by including lagged target variables. By merging these with the exogenous features, we aim to enable the evaluated machine learning models to learn non-stationary components, making it less critical to de-stationarize the data as is commonly required by traditional methods [156].

On the other hand, we want to minimize overhead and the effort required for time

series analysis, feature selection and feature engineering. Further, we want to ensure that the models we examine are applicable in practical settings with limited resources for feature selection and acquisition of exogenous data. In addition, model complexity is a key consideration, and we aim to leverage the inherent ability of recurrent architectures to learn from time-related data without the need for extensive amounts of additional features. We prioritize the time series data itself as the most relevant input to the models, rather than relying on a large number of features that are difficult to obtain in practice, e.g. for energy suppliers or charging station operators.

As a compromise, we decide to exclude meteorological and temperature data and merely include features that can be derived from the original data.

	Energy Demand	**Occupancy**
Learning Task	Regression	Classification
Evaluation Metrics	MASE_{1d}	F1-Score
	RMSE	Accuracy
Baseline Models	Linear Regression	Logistic Regression
Benchmark Models	AdaBoost	
	XGBoost	
Deep Learning Models	PLCnet	
	Conv1D	
	GRU	
	BiLSTM	
	LSTM	
	ConvLSTM	
	CNN-LSTM	
	Stacked LSTM	
	Hybrid LSTM	

Table 4.1: Employed evaluation metrics and models for each learning task.

Overall, nine different deep learning architectures for energy demand forecasting and eight for occupancy forecasting are evaluated. We compare the deep learning models against Linear Regression and Logistic Regression, which serve as respective baseline models. We also evaluate XGBoost and AdaBoost, which serve as easy-to-implement, "off-the-shelf" benchmark models. All employed architectures were discussed (to vary-

ing extents) in the theory part. We employ a seasonally adjusted MASE score to account for the dominant daily seasonal component present in all energy demand time series, as discussed in Section 4.2.3. Table 4.1 provides an overview of the employed models and evaluation metrics for each scenario. Schematic depictions of the layer compositions of all implemented deep learning models are provided in Appendix A.

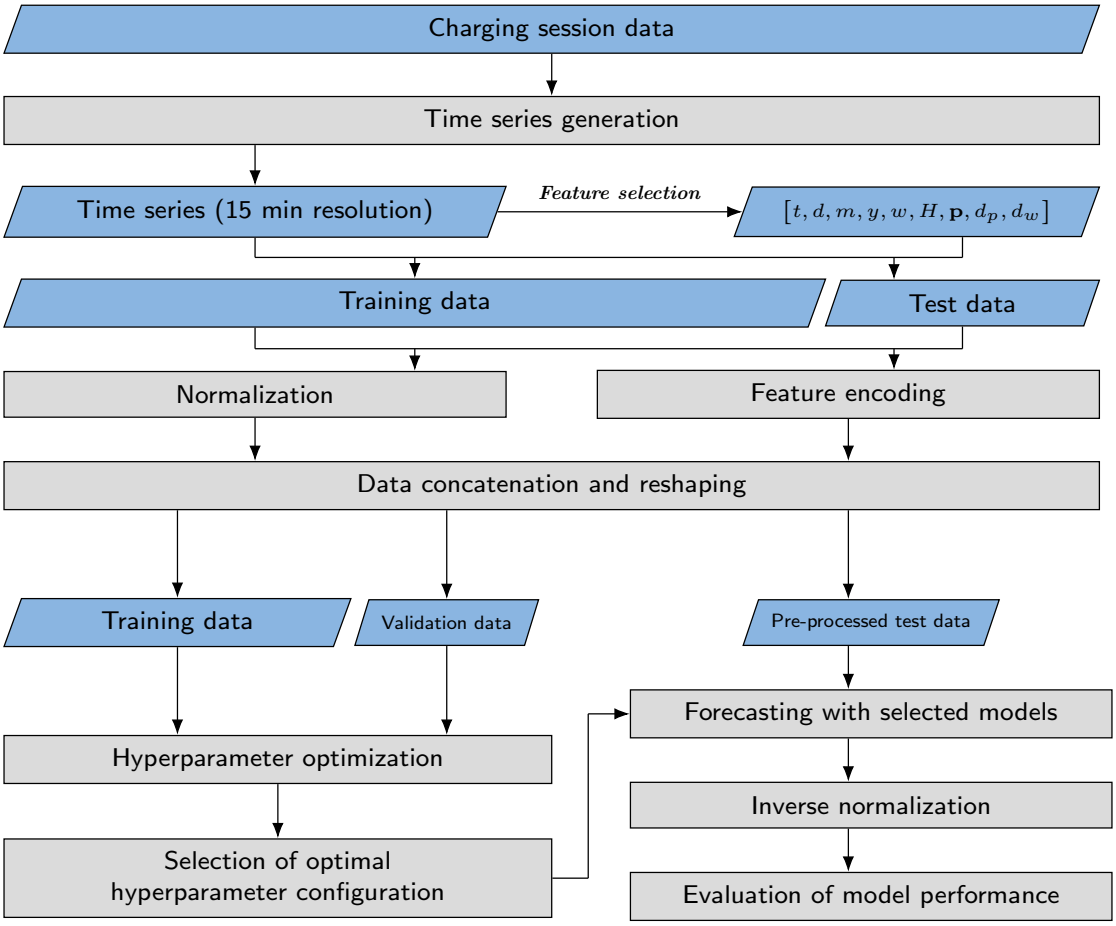


Figure 4.1: Methodical approach for forecasting model development and evaluation.

The evaluation of forecasting models is approached following a three-step procedure:

1. Data pre-processing
2. Model implementation and hyperparameter optimization
3. Time series forecasting and forecast evaluation.

In the following sections, each one of these steps is described in detail, after briefly discussing the chosen of forecasting horizons for each task. Figure 4.1 provides a granular overview of the employed methodical approach.

All models are implemented using `Python 3.10.8`. The libraries `Pandas` and `NumPy` are used for data pre-processing, `scikit-learn` for implementation of baseline and benchmark models and model evaluation. `Keras`, built on a `Tensorflow` backend, is employed for the implementation of deep learning models. Hyperparameter optimization and forecasting are performed using three separate machines:

1. Google Colab Pro+ VM, 53 GB RAM, NVIDIA V100 Tensor Core
2. Intel Xeon E5-2680, 64 GB RAM, NVIDIA GeForce RTX 3060
3. Intel i7-7700HQ, 32 GB RAM, NVIDIA GeForce 940MX.

The main techniques employed to address hardware limitations are the use of the `Keras` mixed precision API and `Keras` data generators, which can significantly reduce memory requirements during model training.

4.1 Forecasting Horizons

Forecasting of EV energy demand and charging station occupancy can be categorized into very short-term (minutes up to hours), short-term (one day up to weeks), medium-term (months up to a year) and long-term (years) [187]. Very short-term and short-term forecasting is used for real-time management of charging stations, while medium-term and long-term forecasting are more useful for planning and decision-making purposes. Longer forecasting horizons commonly require more computational resources and are subject to greater uncertainty. The majority of reviewed existing studies on EV energy demand either perform only single-step predictions or use a very coarse resolution [15, 30, 51, 184, 188, 189, 190, 191], which limits the applicability of the presented models, as energy providers and charging station operators require short-term forecasts for an extended time period in high resolution. This gap is addressed within this thesis by developing models for energy demand forecasting with **1-hour**, **4-hour** and **24-hour** horizons.

The task of predicting EV charging station occupancy requires training a much higher number of models, and it is expected that as the forecasting horizon increases, more complex models are needed, which in turn require more time to train. Additionally, the occupancy scenario is focused on shorter-term forecasts, as most users are primarily interested in knowing whether or not a charging station will be occupied during the next few hours. In light of these considerations, we consider forecasting horizons of **15 minutes**, **1 hour**, and **4 hours** for the EV charging station occupancy scenario.

The MIMO approach is chosen as the multistep forecasting strategy for models based on deep neural networks within this thesis, for three reasons: Its capability to handle complex dependencies between observations, its suitability for large data sets and its straightforward implementation.

The recursive approach is chosen as the multistep forecasting strategy for the baseline models Linear Regression and Logistic Regression as well as for the benchmark models. These architectures lack an inherent multiple-output capability, leading to a preference for the recursive strategy over the direct strategy due to its lower computational costs and a superior capacity to model dependencies between observations.

4.2 Data Pre-Processing

4.2.1 Data Selection

The four source datasets used in this work are sourced from three public repositories: the ACN-Data website [192] and the official websites of the cities of Boulder [193] and Palo Alto [194]. Each dataset comprises a collection of observed EV charging events occurring within a specific geographically-defined area. The properties of the four event-based datasets are provided in Table 4.2.

Dataset	Sector	#CS	#Events	Observation Interval	Occupancy
ACN Caltech	public	51	28,942	05/10/18 - 03/01/20	10.71%
ACN JPL	workplace	52	14,047	10/07/18 - 08/13/19	24.13%
Palo Alto	public	27	103,691	08/01/17 - 03/01/20	22.15%
Boulder	public	20	21,284	01/01/19 - 03/01/20	12.73%

Table 4.2: Properties of the used datasets. #CS denotes the number of individual charging stations, #Events the number of charging events, Occupancy the mean relative occupancy across all charging stations, where the relative occupancy for each charging station is the total occupation time divided by the duration of the full observation interval. The average occupancy rates were calculated after time series generation.

Since deep learning models tend to perform better with larger amounts of data, these particular datasets are chosen due to their high volume of recorded EV charging events. The ACN JPL dataset captures charging events at a workplace charging site accessible

only to employees with key cards, while the ACN Caltech dataset contains events from a single public charging site located in a garage on the Caltech campus. The Boulder and Palo Alto datasets consist of charging events recorded across various public charging stations situated throughout the respective cities.

The datasets are given in a tabular format, where each row corresponds to an individual charging event. The column headers are not standardized across all datasets. The subset of relevant data fields that are consistently present in all datasets includes

1. Charging session ID
2. Charging station ID
3. Connect timestamp (TS_c)
4. Disconnect timestamp (TS_d)
5. Charging duration (Dur_c) in minutes
6. Energy (E) in kWh.

In the first processing step, events occurring at charging stations that were not active at the start of the observation interval are excluded from the respective datasets, for two reasons: to establish a consistent reference timeframe for charging events within each dataset and to avoid unexpected spikes in energy demand that may arise from the introduction of new charging stations within the observed interval. Additionally, any irrelevant columns, i.e. data not required for time series generation, are removed from the datasets. Subsequently, the datasets are cleaned by eliminating rows containing empty, zero, or corrupted entries as well as any erroneous constellations.

Subsequently, we create time series from the event-based datasets.

4.2.2 Time Series Generation

The methods applied for generating time series from the source data depend on the specific forecasting task. In the following, separate sections are dedicated to discussing the respective generation procedures for each task.

4.2.2.1 Energy Demand

The original datasets consist of event-based charging session data from individual charging stations. However, electricity grid operators require energy demand forecasts

on a larger scale, making it impractical to predict demand for each individual charging station. Additionally, training a separate model for each charging station would be computationally expensive. To address these challenges, the data are aggregated across all charging stations within each dataset to create energy demand time series. This approach allows for more efficient forecasting on a dataset level while still capturing the key energy demand patterns across the charging network. The procedure yields four distinct energy demand time series. The ACN JPL time series represents energy demand at workplace charging site, while ACN Caltech represents energy demand at a public charging site. The remaining two time series, Palo Alto and Boulder, represent aggregated energy demand at charging stations on a city-wide level. We choose an interval of 15 minutes as the discrete timestep size. This resolution is a compromise between accuracy and complexity. Also, 15 minutes is the smallest time unit traded on the Intraday Market of EPEX Spot [195].

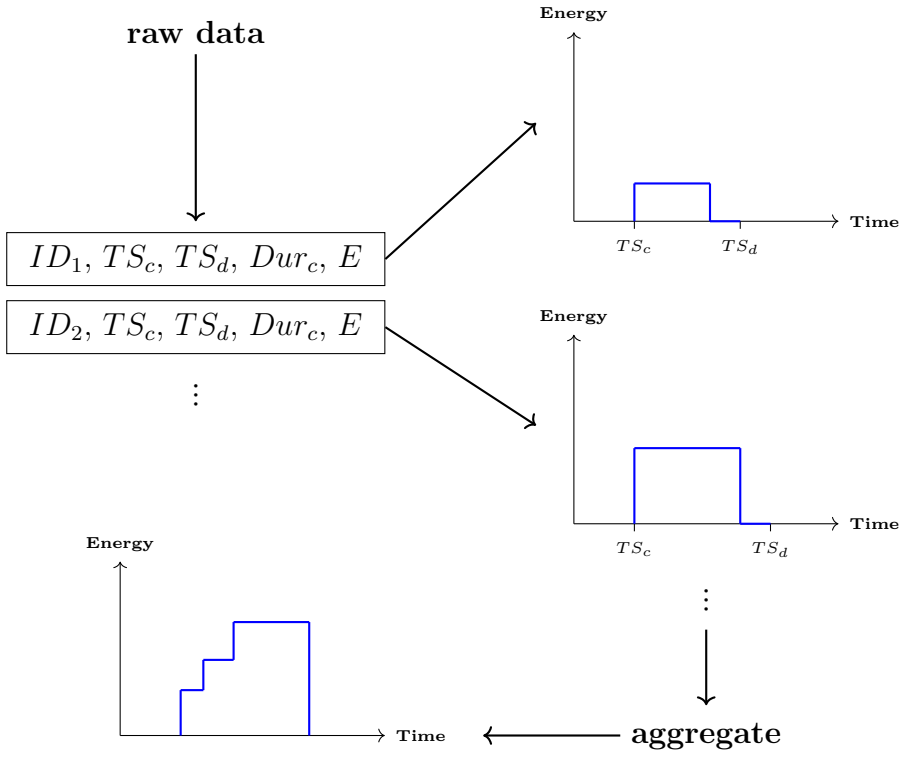


Figure 4.2: Overview of the aggregated energy demand time series generation process.

In order to map the given charging event data to the window size, TS_c and TS_d are rounded to the nearest timestamp representing an integer multiple of 15 minutes and Dur_c is rounded to the next integer multiple of 15. In charging sessions with an initial charging duration below 15 minutes, Dur_c is set to 15. The total session duration

Dur_t is calculated via

$$Dur_t = TS_d - TS_c.$$

Outliers are treated using the approach mentioned in [53] and [196]: Sessions with a total duration Dur_t that is more than three standard deviations from the median in a given dataset are removed. Outliers with respect to total energy consumption E are treated analogously. This results in the removal of an average of 0.80% charging sessions across all data sets.

In the next step, the energy consumed during each 15-minute interval of a charging session, denoted by E_{15}, is calculated via

$$E_{15} = \frac{E}{Dur_c} \cdot 15.$$

Finally, to create one aggregated energy demand time series for each dataset, the charging station data is summed for each 15-minute subinterval of the observation period. Specifically, all E_{15} values that are observed within a given subinterval are added together for all charging stations in the dataset. An overview of the procedure employed for energy demand time series generation is given in Figure 4.2.

4.2.2.2 Occupancy

EV users typically plan their charging schedules around specific charging stations and charging station operators may need to optimize operations or perform maintenance on specific charging stations based on their individual occupancy patterns.

To achieve this granularity, we select an approach corresponding to [53] and [10], resulting in one occupancy state time series for each distinct charging station in each dataset. The resolution is set to 15 minutes, once again representing a compromise between accuracy and complexity. To predict the occupancy status of individual charging stations, the exact timestamp of a future charging session is not relevant and infeasible to predict. Instead, the focus is on the time frame in which a session is expected. The observed variable is discrete in the occupancy case and set to 1 if a charging session occurs within a given interval and to 0 otherwise. To align the charging event data with the desired time intervals, TS_c is rounded down and TS_d is rounded up to the nearest timestamp representing an integer multiple of 15 minutes, and the total session

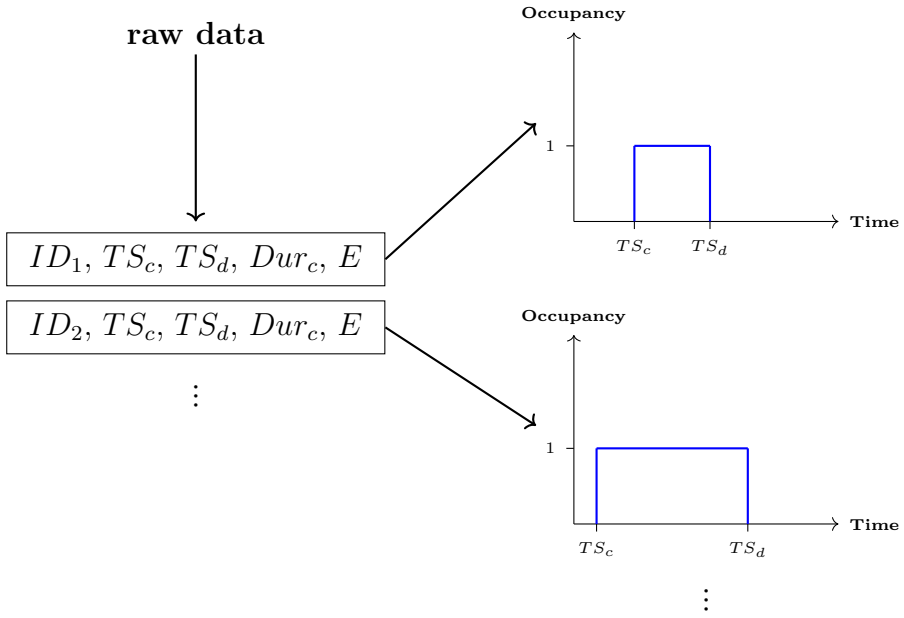


Figure 4.3: Overview of the occupancy state time series generation process.

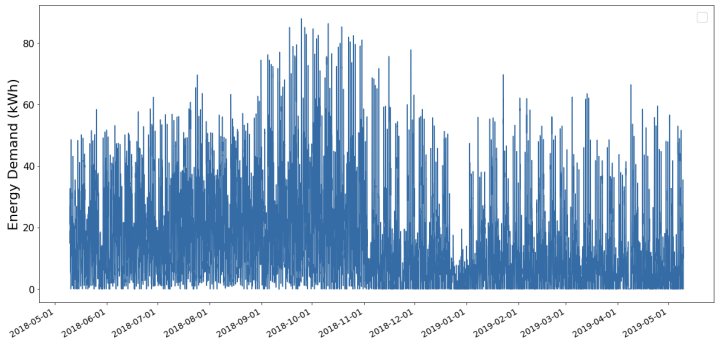
duration Dur_t is calculated via

$$Dur_t = TS_d - TS_c.$$

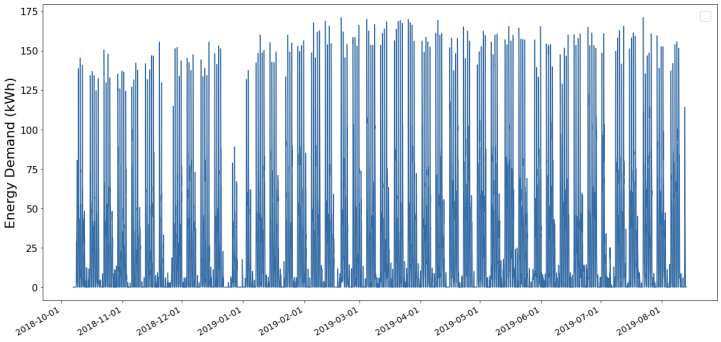
Outliers are treated analogously to the previous section. Subsequently, we generate an occupancy time series for each charging station in each dataset, spanning the total observation interval of the respective dataset, where for each 15 minute subinterval, the observed variable is set to 1 if a charging session took place during this interval, and to 0 otherwise. This procedure results in a total number of 150 distinct occupancy state times series. The procedure employed for occupancy time series generation is illustrated in Figure 4.3.

4.2.3 Time Series Analysis

In order to identify relevant seasonal and trend patterns in the energy demand and occupancy time series and potentially use them to construct additional features, we carry out a brief analysis of the characteristics of the generated time series.



(a) ACN Caltech

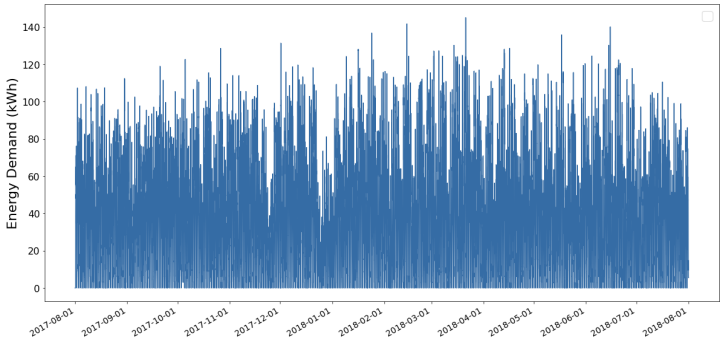


(b) ACN JPL

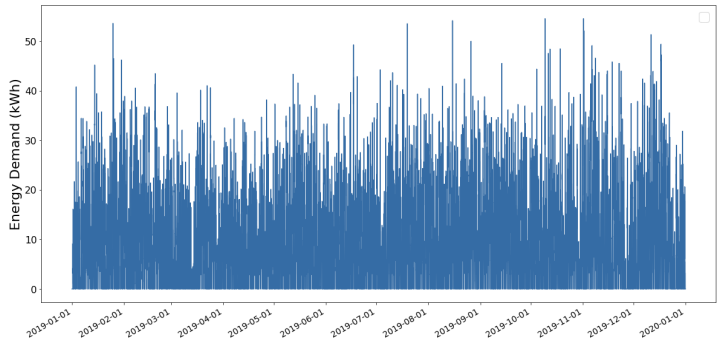
Figure 4.4: Aggregated charging station energy demand time series.

Figures 4.4 and 4.5 depict the aggregated energy demand time series generated from each respective dataset. Several observations can be drawn from the plots. We can observe reduced energy demand during public holiday periods across all datasets. Further, the ACN Caltech curve shows an increasing trend in energy demand during fall, followed by an overall decreasing trend during winter. A slightly increasing trend over time can be observed in the remaining time series, which is likely caused by the overall increase of EV usage throughout the US during the observed intervals [197]. The workplace time series generally shows sharp peak loads with relatively low charging loads during the rest of the day. The ACN Caltech curve also displays high peak loads, but with a more gradual decline in the charging load. The two city-wide time series exhibit an overall lower variance in energy demand throughout the day.

The inspection of each time series across the entire observation interval suggests the presence of both daily and weekly seasonal components, with variations in energy demand depending on the time of day and day of the week. These patterns are particularly prominent in the workplace time series.



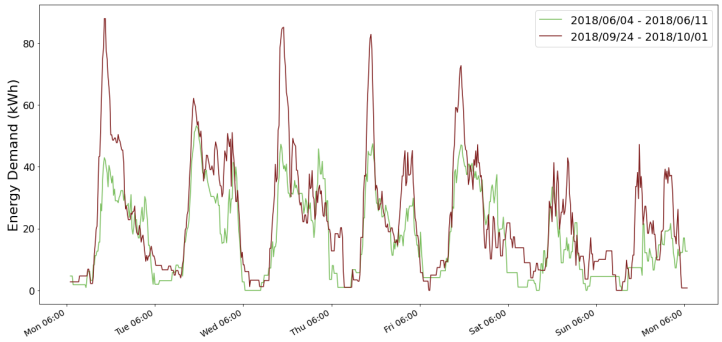
(a) Palo Alto



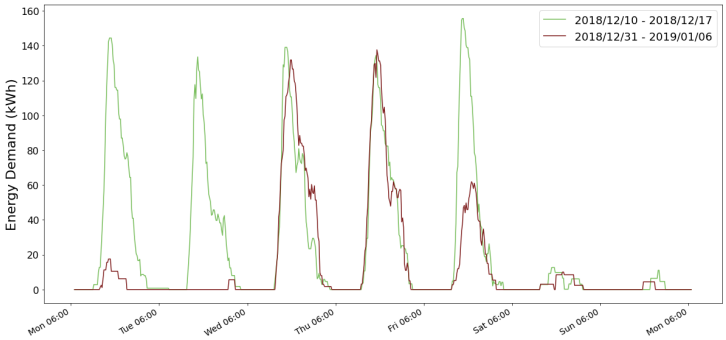
(b) Boulder

Figure 4.5: Aggregated city-wide energy demand time series.

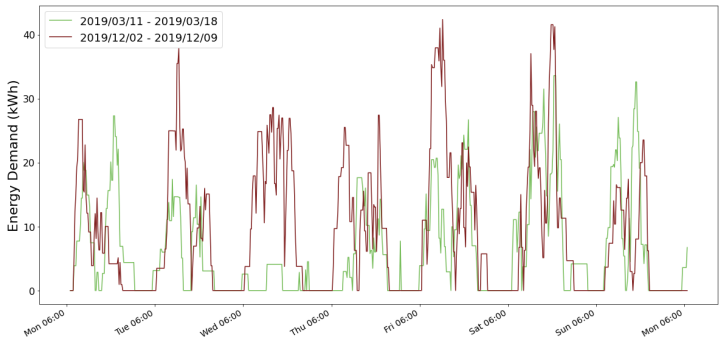
A more detailed analysis of individual weeks, shown in Figure 4.6, can confirm this assumption. When comparing the weekly ACN Caltech energy demand curve in June to that at the end of September, a significant increase in energy demand can be observed. The charging site time series exhibit lower demand during weekends, with the workplace curve showing the most pronounced differences. In contrast, the Boulder time series shows an increase in demand towards the end of the week, suggesting higher mobility of EV users during this time. Across all three plots, energy demand strongly depends on the time of day. On holidays, there is a notable decrease in demand in both the Boulder and the ACN JPL plots.



(a) ACN Caltech



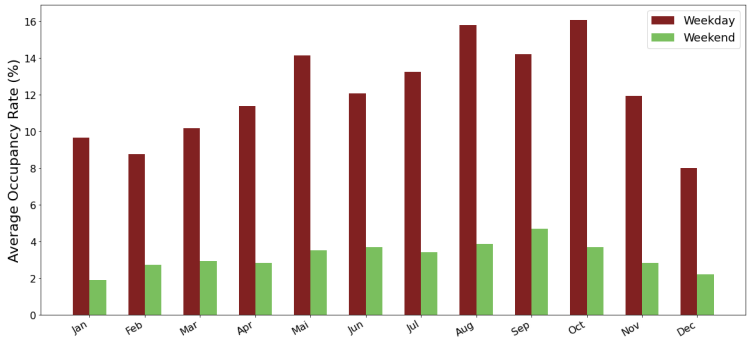
(b) ACN JPL



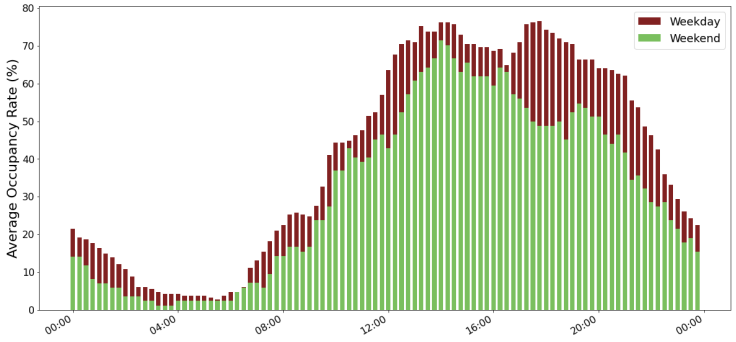
(c) Boulder

Figure 4.6: Energy demand exemplified for different weeks.

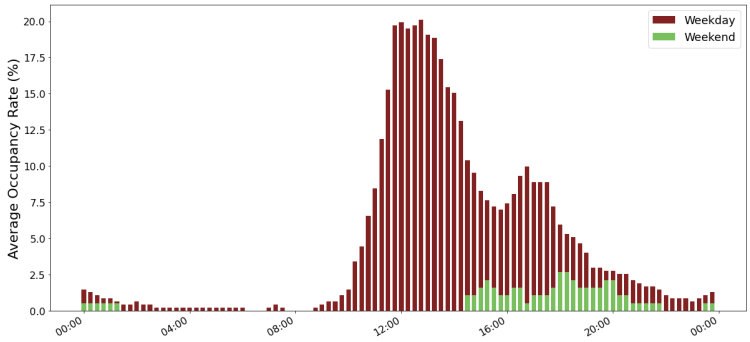
Generating occupancy state time series results in a total number of 150 distinct time series. To analyze meaningful patterns, these series are aggregated for each dataset. Overall, the occupancy state observations exhibit patterns that are very similar to their energy demand counterparts. The respective plots can be found in Appendix D.



(a) ACN Caltech: Average occupancy rates across all charging stations.



(b) Boulder: Average occupancy rates of charging station 1.



(c) Palo Alto: Average occupancy rates of charging station 16.

Figure 4.7: Exemplary average occupancy rate profiles.

By examining the average monthly occupancy rates, shown in Figure 4.7a for the aggregated ACN Caltech time series, it becomes clear that variations exist between individual months. Additionally, we observe that average occupancies fluctuate between weekdays and weekends. The remaining plots in Figure 4.7 imply that the occupancy state is highly dependent on the time of day and the type of weekday. By comparing

the plots of the average occupancy rates of Boulder CS1 and Palo Alto CS16 in Figure 4.7, we get an impression of the amount of variation between occupancy state profiles within the city-wide time series.

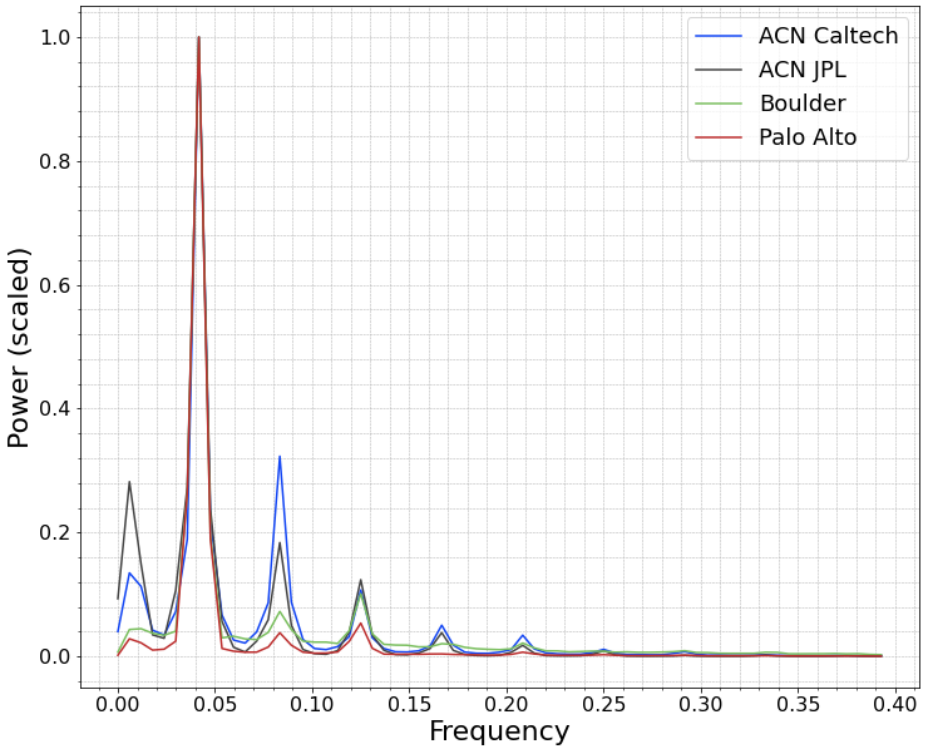


Figure 4.8: Spectrograms of the energy demand time series, the frequency is displayed in h^{-1}. The city-wide time series exhibit a more sparse frequency domain representation.

To further identify seasonal patterns in the energy demand time series, we create frequency spectrograms using the `scipy.fft` library, employing a Hamming window of one week. The peaks in the resulting spectrograms, displayed in Figure 4.8, confirm the previously discussed observations. All time series display a prominent daily seasonal pattern. Additionally, the charging site series exhibit pronounced half-daily and weekly cycles that are less distinct in the city-wide series. Moreover, 8-hour cycles are evident across all time series, while 6-hour cycles are only present in the charging site time series. Overall, the charging site series, which aggregate observations from multiple charging stations in close proximity, exhibit more clearly discerned periodic patterns in their spectrograms, which can be attributed to more synchronized usage patterns of the charging stations.

4.2.4 Feature Selection

Based on the results of the time series analysis presented earlier and guided by the reasoning outlined at the beginning of this chapter, we enhance the feature set for both energy demand and occupancy state forecasting tasks by incorporating exogenous, time-related features. As the energy demand and occupancy state time series were derived from the same original datasets and exhibit similar characteristics when aggregated, we use the same exogenous features for both tasks. This approach also aims to minimize the effort required for feature engineering. Table 4.3 depicts the time-related features that are included in the feature sets for both energy demand and occupancy state forecasting. The concept of cyclic feature encoding is discussed in the next section.

Feature Name	**Variable**	**Value Range**	**Cyclic Encoding**
Time of day	t	$\{0, ..., 95\}$	yes
Day of week	d	$\{0, ..., 6\}$	yes
Day of month	m	$\{0, ..., 30\}$	yes
Day of year	y	$\{0, ..., 364\}$	yes
Weekend/weekday	w	$\{0, 1\}$	no
Holiday	H	$\{0, 1\}$	no

Table 4.3: Included time-related exogenous features for both forecasting tasks.

Additionally, we incorporate a feature vector that represents the long-term tendencies of the respective time series. This approach has been shown to improve the performance of forecasters in [53]. It allows the models to capture patterns that are not explicitly present in the shorter-term variations, thus enabling them to extrapolate future observations based on the expected trends of the data over a longer horizon. To this means, for the energy demand time series, two vectors containing the average energy demand for each 15-minute interval of the day are generated from the respective training datasets after normalization (see Section 4.2.6), where the first vector contains the weekday averages and the second vector the weekend averages, taken across the entire observation interval of the respective training dataset. This is done analogously for the occupancy time series, resulting in two vectors per charging station, consisting of its respective weekday and weekend occupancy probabilities for each 15-minute interval of the day. The historical profile vectors generated from the training data are also used for the test dataset.

Feature Name	Variable	Range (En. ‖ Oc.)	Meaning
Historical profile	$\mathbf{p}$	$\mathbb{R}^{96} \parallel [0,1]^{96}$	Long-term tendencies
Previous day	d_p	$\mathbb{R} \parallel \{0,1\}$	Observation at $t-96$
Previous week	d_w	$\mathbb{R} \parallel \{0,1\}$	Observation at $t-672$

Table 4.4: Features capturing long-term tendencies and lagged variables.

To further enhance the models' ability to learn non-stationary patterns in the data, we incorporate two additional lagged variables as features. The daily seasonal component is captured by the observation taken precisely 24 hours prior to the current time step, whereas the weekly seasonal component is represented by the observation from 7 days earlier. Features that aim at capturing long-term tendencies and the lagged variables are summarized in Table 4.4.

In the next step of data preprocessing, we encode selected features using *cyclic feature encoding*.

4.2.5 Feature Encoding

One of the most commonly used methods to represent categorical data for machine learning algorithms is one-hot encoding, which transforms categorical features into a binary vector representation. Given a categorical feature x that can take K unique values $c_1, c_2, \ldots, c_K$, one-hot encoding is performed by creating K binary features $x_1, x_2, \ldots, x_K$, where:

$$x_i = \begin{cases} 1, & \text{if } x_i = c_i \\ 0, & \text{otherwise.} \end{cases}$$

This ensures that each category is represented as a distinct and independent feature. However, when it comes to cyclical features like hour of the day or day of the week, one-hot encoding results in a large number of sparse binary vectors with jump discontinuities that do not capture the inherent cyclical nature of the data [198].

This issue is addressed by a method called cyclic feature encoding. Cyclic encoding transforms a given cyclical feature into two new features which capture the periodicity of the original feature. The procedure can improve the performance of deep learning

models by allowing them to capture cyclical patterns in the data. The authors in [199] found that cyclic encoding is potentially more effective than one-hot encoding for time series forecasting tasks.

Using a method defined in [200], cyclic feature encoding within this work is realized by splitting features into a cosine and sine part. Given a categorical feature x that takes values between 0 and K, where K is the period of the cycle, the method calculates

$$x_{sin} = \sin\left(\frac{2\pi x}{K}\right),$$

$$x_{cos} = \cos\left(\frac{2\pi x}{K}\right).$$

The resulting encoded feature is then given by $[x_{sin}, x_{cos}]$, where $[\ \cdot\ ,\ \cdot\ ]$ denotes concatenation.

4.2.6 Data Normalization

It is common practice to perform data normalization for machine learning for several reasons. It improves numerical stability during training as a large variance in feature scales can lead to vanishing or exploding gradients. It can also accelerate optimizer convergence during training and improve the generalization ability of a model as it will be less affected by the scale of the input features [201, 202].

The real-valued energy demand time series is normalized by performing *min-max-normalization*, which is a form of feature scaling and a commonly used technique for time series data [15, 51, 203].

A mistake frequently made in machine learning is the encoding and pre-processing of data before dividing it into sets for training, validation, and testing, which can lead to information leakage from the training to the test dataset [204].

Keeping the previous aspects in mind, we initially split the generated time series into training and test datasets, using a ratio of $\mathbf{0.75}$ to $\mathbf{0.25}$. The occupancy time series are split according to the same ratio. However, data normalization is not necessary, as the observed values and average occupancy probabilities are within $[0, 1]$. The energy

demand time series are normalized in the range $[0, 1]$ by applying min-max scaling:

$$E_{t,norm} = \frac{E_t - E_{min}}{E_{max} - E_{min}},$$

where $E_{t,norm}$ is the normalized energy demand and E_t is the actual energy demand at time t. E_{max} and E_{min} are the minimum and maximum energy demand values of the respective time series. Normalization for training and test data is performed separately to avoid data leakage. In the first step, training data is normalized. Next, the test sets are normalized using the minimum and maximum values obtained from the corresponding training datasets.

4.2.7 Data Reshaping

The last data pre-processing step comprises the reshaping of the time series and corresponding feature vectors in order to obtain datasets suited as input for the various models. We have to consider two distinct approaches here, depending on the architecture at hand. Models without sequential data processing capabilities, i.e. Linear and Logistic Regression as well as AdaBoost and XGBoost, rely on datasets generated by a time delay embedding procedure as defined in Section 3.2.1. Given a univariate times series $(x_1, ..., x_\tau, ..., x_T)$ of either energy demand observations or the occupancy sequence of a single charging station with corresponding feature vectors $(\mathbf{f}_1, ..., \mathbf{f}_\tau, ..., \mathbf{f}_T)$ and a window size p, the employed data reshaping procedure generates training and test data sets of the following structure:

$$\underbrace{\begin{array}{l} \textbf{Features} \\ \textbf{Labels} \end{array} \left\{ \begin{bmatrix} f_{p+1,1} & f_{p+2,1} & \dots & f_{\tau-1,1} \\ f_{p+1,2} & \ddots & & f_{\tau-1,2} \\ \vdots & & \ddots & \vdots \\ f_{p+1,m} & f_{p+2,m} & \dots & f_{\tau-1,m} \\ x_1 & x_2 & \dots & x_{\tau-p-1} \\ x_2 & \ddots & & x_{\tau-p} \\ \vdots & & \ddots & \vdots \\ x_{p+1} & x_{p+2} & \dots & x_{\tau-1} \\ x_{p+2} & x_{p+3} & \dots & x_\tau \end{bmatrix} \right.}_{\textbf{Training Data}} , \quad \underbrace{\begin{bmatrix} f_{\tau,1} & f_{\tau+1,1} & \dots & f_{T-1,1} \\ f_{\tau,2} & \ddots & & f_{T-1,2} \\ \vdots & & \ddots & \vdots \\ f_{\tau,m} & f_{\tau+1,m} & \dots & f_{T-1,m} \\ x_{\tau-p} & x_{\tau-p+1} & \dots & x_{T-p-1} \\ x_{\tau-p+1} & \ddots & & x_{T-p} \\ \vdots & & \ddots & \vdots \\ x_\tau & x_{\tau+1} & \dots & x_{T-1} \\ x_{\tau+1} & x_{\tau+2} & \dots & x_T \end{bmatrix}}_{\textbf{Test Data}}$$

The window size p is a hyperparameter, and multiple versions of training/test datasets have to be generated, one for each possible value of p, as shown in Table 4.5.

The remaining models implement the MIMO strategy and process sequences, following a *stateless* approach with a stride length of one: For a fixed window size p and forecasting horizon h, at each time step t, the single-stream models process p feature vectors $\mathbf{x}_{t-p}, ..., \mathbf{x}_t$, and receive one label $\mathbf{y}_t$, where

$$\mathbf{x}_j = \begin{bmatrix} f_{j,1} \\ f_{j,2} \\ \vdots \\ f_{j,m} \\ x_j \end{bmatrix}, \quad j = t-p, ..., t \qquad \text{and} \qquad \mathbf{y}_t = \begin{bmatrix} x_{t+1} \\ x_{t+2} \\ \vdots \\ x_{t+h} \end{bmatrix}. \tag{4.1}$$

The Hybrid LSTM model processes the original time series data and the additional features in separate paths and its inputs at each time step t have the shape

$$\underbrace{\begin{bmatrix} x_j \end{bmatrix}}_{\textbf{LSTM}}, \quad \underbrace{\begin{bmatrix} f_{j,1} \\ f_{j,2} \\ \vdots \\ f_{j,m} \end{bmatrix}}_{\textbf{FFN}}, \quad j = t-p, ..., t.$$

Finally, at each timestep t, the PLCnet model uses two identical copies of $\mathbf{x}_j$ as specified in Equation 4.1 as inputs for its LSTM and CNN paths, respectively. The labels for the multi-stream models have the same structure as given in Equations 4.1. For sequential processing, only one training/test dataset is created that can be used for different values of the hyperparameter p.

Before hyperparameter optimization can start, we divide each training dataset into a final training dataset set and a hold-out validation set, using a ratio of **0.75** to **0.25**.

4.3 Hyperparameter Optimization

As mentioned in Section 2.3.7, hyperparameters are not directly estimated from the training data. They are external to the model and typically set before learning starts, although some hyperparameters can be tweaked during training. Hyperparameter tuning is the process of sampling model architecture candidates from the space of possible hyperparameter values. This is often referred to as "searching" the hyperparameter space for optimum values. Finding optimal hyperparameters is generally a time-consuming iterative procedure. There are three commonly used methods for tuning hyperparameters in machine learning: manual search, grid search, and random search. While these methods are straightforward to implement, more advanced techniques such as Bayesian optimization exist but are not discussed within this work due to their complexity [205, 206].

Manual search involves selecting hyperparameter settings manually based on prior knowledge and intuition. Grid search involves specifying a subset of the hyperparameter space of a machine learning algorithm and searching for the optimal combination of hyperparameters using an exhaustive approach. Random search is a probabilistic search algorithm that randomly samples hyperparameter values from a pre-defined distribution over the hyperparameter space.

The approach used in this thesis for finding optimal hyperparameter configurations follows the findings presented by Bergstra and Bengio in [207]. The authors provide empirical evidence that random search, conducted over the same domain as a grid search approach, can find models that are as good as, or better than, those found by grid search in a fraction of the computation time. In one of their experiments, Bergstra and Bengio considered a 32-dimensional hyperparameter space for a neural network model and found that purely random search found statistically equal performance on four out of seven datasets and superior performance on one dataset when compared to a thoughtful combination of manual and grid search. They also showed analytically that, for the generalization performance of models on most of the considered datasets, only a small number of hyperparameters really matter, and that different hyperparameters are important on different datasets. Based on these findings, the authors concluded that grid search is a poor choice, especially when configuring algorithms for new datasets. Instead, they recommend using random search to explore the hyperparameter space more efficiently and thus allowing for the possibility of finding good or even optimal hyperparameters.

Adhering to this rationale, we choose random search for hyperparameter tuning. This approach compensates for lack of experience in manual hyperparameter selection and effectively explores the hyperparameter space, given the considerable number of models that must be trained on a considerable number of heterogeneous datasets.

If we let $\boldsymbol{\lambda}_i \in \boldsymbol{\Lambda}$ denote the hyperparameter configuration evaluated at iteration i of the search algorithm, where $\boldsymbol{\Lambda}$ denotes the hyperparameter space, estimating the optimal hyperparameter configuration can be written as

$$\begin{aligned}\boldsymbol{\lambda}^* &= \underset{\boldsymbol{\lambda}\in\boldsymbol{\Lambda}}{\operatorname{argmin}} \sum_{i=1}^{N_{val}} \mathcal{L}(\hat{f}_{\boldsymbol{\lambda}}(\mathbf{x}_i), \mathbf{y}_i) \\ &\approx \underset{\boldsymbol{\lambda}\in\{\boldsymbol{\lambda}_1,\ldots,\boldsymbol{\lambda}_k\}}{\operatorname{argmin}} \sum_{i=1}^{N_{val}} \mathcal{L}(\hat{f}_{\boldsymbol{\lambda}}(\mathbf{x}_i), \mathbf{y}_i) \equiv \hat{\boldsymbol{\lambda}},\end{aligned}$$

i.e. the objective is to minimize the validation loss. For a fixed hyperparameter space $\boldsymbol{\Lambda}$, finding a configuration among the top 5% of all possible configurations with a probability of 95% takes random search less than 60 trials.

To fully utilize random search, high-dimensional hyperparameter spaces are used for all deep learning architectures considered in this thesis. To evaluate whether boosting methods can achieve similar performance with less configurational overhead and an overall lower computational cost, their respective hyperparameter spaces are chosen to have lower dimensions.

4.3.1 Hyperparameter Spaces

4.3.1.1 Baseline

Apart from the window size p, no hyperparameter optimization is carried out for the baseline models. The optimal window is determined using a standard grid search approach across all forecasting horizons and datasets, where the value range for p is set to $\{2, 4, 8, 16, 32\}$, and the total number of models trained on the occupancy time series is reduced by following the approach presented towards the end of Section 4.3.1.2, with the difference that distinct Logistic Regression models are trained for each forecasting horizon. Additionally, the Linear Regression models use L_2-regularization (Ridge regression), using the default `scikit-learn` setting of $\alpha = 1$.

4.3.1.2 Deep Learning

A number of settings are shared across models and forecasting tasks. `Keras` standard settings are used for weight and bias initialization for all examined deep learning models. Recurrent layers use their standard activation function hyperbolic tangent, while hidden, dense and convolutional layers employ ReLu. Dense output layers use sigmoid as activation. Convolutional layers use a fixed kernel size of 4 while maximum pooling layers use a window of size 2. To preserve temporal dependencies, training data is not shuffled during the learning process. In general, settings not mentioned in the following sections use the `Keras` or `scikit-learn` standard, respectively.

Regression models aim to optimize the mean squared error loss function:

$$\mathcal{L}_{MSE} = \frac{1}{T}\sum_{i=1}^{T} \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2,$$

while classifiers employ logistic loss, also referred to as binary cross-entropy, as their loss function:

$$\mathcal{L}_{BCE} = \frac{1}{T}\sum_{i=1}^{T} \left(-y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right).$$

Regarding the lookback window size p, choosing sizes beyond approximately 16 steps is not feasible for some deep learning models when dealing with a forecasting horizon of $h = 96$, due to memory constraints. As a consequence, the maximum lookback window size is limited to $p = 16$ time steps.

Hyperparameter	**Range**	**Default**
Optimizer	{ADAM, SGD}	ADAM
Clip Norm	{None, 1.1, 2.0, 4.0, 8.0}	None
Batch Size	{4, 8, 16, 32, 64, 128}	32
Number of Epochs	{16, 32, 64, 128, 256}	10
Window Size p	{2, 4, 8, 16}	10

Table 4.5: Hyperparameters shared across all deep learning architectures. The number of epochs is additionally regulated by an early stopping criterion, where the patience is set to $\frac{n\_epochs}{8}$. *Clip Norm* is the threshold ξ from Section 3.4.2.2. Keras standard settings are used for the learning rates of the respective optimizers. SGD is initialized with momentum $m = 0.9$.

The hyperparameter ranges presented within this section are independent of the forecasting task, i.e. the same hyperparameter space is used for both energy demand forecasting and occupancy forecasting models. Hyperparameter values are sampled from their respective ranges uniformly at each iteration of random search.

Table 4.5 displays hyperparameter ranges that are shared across all examined deep learning architectures, while hyperparameters governing the architectural composition of specific models are presented in Table 4.6.

Hyperparameter	**Range**	**Model**
Include Convolution 1	{True, False}	Hybrid LSTM
Stack Layers	{1, 2, 3}	Hybrid LSTM
Include Convolution 2	{True, False}	PLCnet
Include Dense	{True, False}	PLCnet
Stack Size	{2, 3, 4}	Stacked LSTM
Stack Dilate	{3, 4} × {0, 1}	Conv1D

Table 4.6: Hyperparamters governing layer composition of models. *Include Convolution* replaces the FFN module at the beginning of the model with a dense layer followed by two convolutional layers, one maxpooling and one flatten layer. *Stack Layers* controls how many hidden layers are stacked in the LSTM module. *Include Convolution*, if set to False, replaces the CNN module at the beginning of PLCnet by two fully connected layers followed by a dropout layer. *Include Dense* adds a fully connected layer at the end of the CNN path, before its output is concatenated with the LSTM module's output. *Stack Size* controls how many hidden LSTM layers are stacked. The first entry of *Stack Dilate* controls how many hidden convolutional layers are stacked (default = 2), the second entry applies a dilation rate of 2^i to hidden layer i if set to 1.

The architectural composition of the following models is not influenced by hyperparameters:

- LSTM
- BiLSTM
- CNN-LSTM
- CNN-GRU
- GRU
- 1D-CNN
- CNN-BiLSTM
- ConvLSTM.

Table 4.7 is similar to Table 4.5; it displays value ranges of hyperparameters that are shared across all deep learning models. The listed hyperparameters govern the composition and the behavior of individual layers, if applicable.

Hyperparameter	**Range**	**Default**
Filters 1	{4, 8, 16, 32, 64}	32
Filters 2	{4, 8, 16, 32, 64}	32
Nodes Dense 1	{8, 16, 32, 64, 128, 256}	1
Nodes Dense 2	{8, 16, 32, 64, 128}	1
Nodes Dense 3	{8, 16, 32, 64, 128}	1
Nodes Dense 4	{8, 16, 32, 64, 128}	1
Nodes Recurrent 1	{8, 16, 32, 64, 128, 256}	128
Nodes Recurrent 2	{8, 16, 32, 64, 128}	128
Nodes Recurrent 3	{8, 16, 32, 64, 128}	128
Nodes Recurrent 4	{8, 16, 32, 64, 128}	128
Weight Regularization 1	{None, None, None, L_1, L_2}	None
Weight Regularization 2	{None, None, None, L_1, L_2}	None
Weight Regularization 3	{None, None, None, L_1, L_2}	None
Weight Regularization 4	{None, None, None, L_1, L_2}	None
Dropout 1	$[0.01, 0.6]$	0
Dropout 2	$[0.01, 0.6]$	0

Table 4.7: The *Filters i* hyperparameter controls the number of kernels in the i-th convolutional layer. The *Nodes Dense i* and *Nodes Recurrent i* parameters determine the number of units in the i-th dense or dimension of the hidden state of the i-th recurrent layer, respectively. *Weight Regularization i* controls the kind of weight penalty applied during training in dense layer i. *Dropout i* defines which portion of activations dropout layer i are set to zero during training.

The approach to hyperparameter optimization varies depending on the specific forecasting task. For energy demand forecasting, where models only have to be evaluated on four different time series, random search is conducted with $k = 30$ trials for each model, dataset, and forecasting horizon.

This method is not feasible for occupancy forecasting, given the computational and time constraints. Instead, we opt to train models only for a forecasting horizon of $h = 8$, and later evaluate models using the respective optimal hyperparameter configurations on horizons $h \in \{1, 4, 16\}$. Additionally, rather than testing each hyperparameter con-

figuration on all available occupancy time series, we limit the number of time series included to 10 for each site, where site refers to the respective original datasets (i.e. there are 4 sites in total). This selection always includes the two charging stations with the highest and lowest average occupancy rates, for each site respectively. The remaining eight series for each site are chosen uniformly at random during each trial of random search, which again runs for use $k = 30$ trials for each model. By employing an early stopping criterion and distributing the process across three machines, we can reasonably enhance the model training throughput during hyperparameter optimization.

4.3.1.3 Benchmark

Hyperparameter	**Range**	**Default**
Number of Estimators	{32, 64, 128}	50
Model Depth	{1, 2, 3}	1
Learning Rate	{0.1, 0.5, 1.0, 2.0}	1.0

Table 4.8: *Number of Estimators* denotes the number of iterations the algorithm will run (number of weak learners, M in Algorithm 3). *Model Depth* is an upper bound for the depth of weak learners. For example, when set to 2, decision trees will have a maximum of 4 leaves.

AdaBoost has a number of hyperparameters than can be tweaked in order to find an optimal model configuration for the task at hand. In the `scikit-learn` implementation that was used within this thesis, the default setting for the underlying weak learner used to perform classification is a simple decision tree classifier and a decision tree regressor for regression tasks. Table 4.8 lists the ranges of hyperparameters which were considered for optimizing AdaBoost's performance within this work. The learning rate η, which is implemented by default in `scikit-learn`, scales each weak learner before it is added to the ensemble:

$$\hat{f}_m(\mathbf{x}) = \eta\hat{\theta}_m\hat{\phi}_m(\mathbf{x}).$$

Hyperparameter	**Range**	**Default**
Number of Estimators	{32, 128, 512}	100
Maximum Depth	{3, 6, 10}	6
Learning Rate	{0.025, 0.1, 0.4}	0.3
Alpha	{0, 0.5, 1}	0
Lambda	{0.5, 1, 2}	1

Table 4.9: *Number of Estimators* denotes the number of iterations the algorithm will run (number of weak learners, M in Algorithm 4). *Maximum Depth* provides an upper bound for the depth of the individual decision tree classifiers or regressors, respectively. *Learning Rate* (η in Algorithm 4) scales each weak learner before it is added to the ensemble.

The `scikit-learn` implementation of XGBoost provides a large number of hyperparameters that can be tuned to increase the predictive performance of the algorithm. Table 4.9 displays the ranges of hyperparameters that were considered for optimizing the algorithm's performance within this work. By default, the complexity of individual trees is penalized via

$$\Omega(f) = \sum_{m=1}^{M} \left(\gamma T_m + \frac{1}{2} \lambda \|w_m\|^2 + \alpha \|w_m\|_1 \right)$$

in the `scikit-learn` implementation.

As for deep learning models, random search is run for $k = 30$ trials in the case of boosting models. To limit the number of models that must be trained on the occupancy state time series, we adopt a similar approach to the one outlined in the previous section. The key difference is that we train distinct models for each forecasting horizon.

Chapter 5

Results

The results of hyperparameter optimization and forecasting evaluation are discussed in separate sections for each forecasting task, starting with the energy demand scenario.

5.1 Energy Demand

5.1.1 Hyperparameters

Due to the extensive range of models and possible configurations, the overall results of hyperparameter optimization present a highly variable landscape, making the identification of discerning patterns challenging. Also, due to the time constraint, a detailed analysis of hyperparameter optimization results is omitted.

Instead of listing all optimal configurations explicitly, we will focus on discernible trends and on the models that ultimately showed the best generalization ability during model evaluation.

The findings suggest that, as the forecasting horizon increases, models of higher complexity tend to perform better, which can mostly be attributed to a trend towards a larger number of units in the recurrent layers and a larger preferred lookback window size p. When averaging across models and datasets, the optimal hyperparameter configurations exhibit the following combinations of parameter counts, lookback window sizes, and average dropout rates, for each respective forecasting horizon:

- 1 hour: 92,095 | 8.1 | 0.26
- 4 hours: 136,219 | 8.8 | 0.17
- 24 hours: 198,837 | 12.3 | 0.25.

Generally, the models that exhibit the best performance on the ACN time series have a parameter count that is 14% larger than the models performing best on the city-wide series. This is likely due to the stronger presence of seasonal patterns in the ACN observations, which require a larger number of parameters to capture effectively.

Another prominent observation is that the vast majority of deep learning models achieving minimal validation loss rely on ADAM as their optimizer. Overall, only 9% of models with the best-performing respective hyperparameter configurations use SGD, averaged across all datasets and forecasting horizons.

Regarding batch size and the number of epochs used for training, no clear patterns can be discerned. The application of early stopping further obscures the landscape. The same applies to the hyperparameter ξ and the overall patterns of applied L_1- and L_2-regularization to weights in the dense layers.

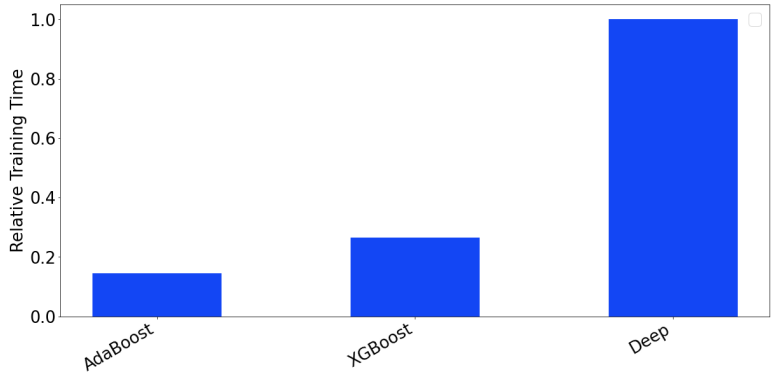


Figure 5.1: Comparison of the relative training times observed for boosting models and deep learning models during the hyperparameter optimization process for energy demand forecasting, averaged across forecasting horizons and datasets.

Concerning the hyperparameters that determine the architectural composition of the multi-stream models, the Hybrid LSTM model appears to benefit from stacking layers in its LSTM module. On average, the best models trained for 4-step forecasts exhibit a stack size of 1.25, while the best models for 16 and 96-step forecasts each have an average of 1.75 stacked layers. The model does not seem to gain a significant advantage from replacing the fully connected layer input stream with a convolutional layer, as only one out of 12 selected models exhibits this feature.

Regarding PLCnet, 10 out of 12 selected models exhibit a combination of the original CNN input path with an additional fully connected layer added at the end of the path, before concatenation.

In general, the benchmark models require only a fraction of the training time needed for deep architectures when tackling the same task, as shown in 5.1, while their overall implementation is noticeably less complex.

1 hour ‖ 4 hours ‖ 24 hours	ACN Caltech	ACN JPL	Palo Alto	Boulder
Window Size p	8 - 8 - 16	8 - 4 - 8	4 - 8 - 16	4 - 16 - 8
Number of Estimators	512 - 512 - 128	512 - 128 - 128	128 - 128 - 128	128 - 32 - 512
Maximum Depth	3 - 3 - 6	3 - 6 - 6	3 - 3 - 6	3 - 6 - 3
Learning Rate	0.1 - 0.1 - 0.025	0.025 - 0.1 - 0.4	0.1 - 0.4 - 0.1	0.025 - 0.025 - 0.025
Alpha	0.5 - 0 - 1.0	0 - 0.5 - 0.5	0.5 - 0 - 0	0 - 0 - 1.0
Lambda	0 - 0.5 - 1.0	1.0 - 0.5 - 2.0	0.5 - 0.5 - 1.0	1.0 - 1.0 - 1.0

Table 5.1: Hyperparameter configurations of the overall best-performing models in four-step ahead energy demand forecasting. Configurations for 16-step and 96-step ahead forecasts are included for a more comprehensive analysis.

4 hours ‖ 24 hours	ACN Caltech	ACN JPL	Palo Alto	Boulder
Window Size p	8 - 8	4 - 16	16 8	2 8
Optimizer	ADAM - ADAM	ADAM - ADAM	ADAM - ADAM	ADAM - ADAM
Clip Norm	1.1 - 4.0	2.0 - 1.1	2.0 - 4.0	8.0 - 4.0
Batch Size	128 - 64	64 - 32	32 - 64	64 - 32
Epochs	128 - 128	16 - 64	32 - 32	16 - 32
Filters 1	32 - 4	8 - 4	16 -	32 -
Filters 2	8 - 8	4 - 16	4 -	8 -
Nodes Dense 1	8 - 64	32 - 16	128 - 16	8 - 64
Nodes Rec. 1	128 - 128	64 - 128	32 - 64	32 - 32
Weight Reg. 1	None - None	L_2 - L_2	None - None	None - None
Weight Reg. 2	None - L_2	None - None	None - None	L_2 - None
Weight Reg. 3	None - None	None - L_1	L_1 - None	None - L_1
Dropout 1	0.17 - 0.21	0.23 - 0.09	0.20 - 0.15	0.41 - 0.34

Table 5.2: Hyperparameter configurations of the overall best-performing models at 16 and 96 steps-ahead energy demand forecasting. The left entries in the columns represent the respective configuration or 16-step forecasts, the right entries are the hyperparameter of models trained for 96-step forecasts. The models displayed per column are, from left to right: (CNN-LSTM ‖ CNN-LSTM), (CNN-LSTM ‖ CNN-LSTM), (CNN-LSTM ‖ BiLSTM), (CNN-LSTM ‖ BiLSTM).

For a 1-hour horizon, XGBoost demonstrated the best overall forecasting performance, with corresponding hyperparameter configurations displayed in Table 5.1. Additionally, we have included hyperparameter settings for other horizons to better understand XGBoost's comparatively weaker performance when forecasting at larger horizons. A noticeable trend is a preference for a number of base estimators of at least

128, which display relatively low individual complexity. Furthermore, XGBoost tends to profit from larger lookback window sizes. In general, increased regularization tends to improve the model's performance as the forecasting horizon expands.

When examining the longer forecasting horizons of 4 hours and 24 hours, the models exhibiting the best overall forecasting performance during model evaluation are CNN-LSTM and BiLSTM. Their respective hyperparameter configurations can be found in Table 5.2. The ACN time series tend to require models of higher overall complexity for accurate forecasts. The ADAM optimizer is predominant, and the overall model complexity exhibits an increasing trend as the forecasting horizon grows. Gradient clipping is applied across all models, with varying threshold values for ξ. The weight regularization parameters and the number of filters in the convolutional layer do not exhibit clearly discernible patterns.

In the subsequent step, models that attained mimimum validation loss during hyperparameter optimization are assessed by performing energy demand forecasts. The results of this evaluation are presented in the following section.

5.1.2 Model Evaluation

After determining favorable hyperparameter configurations, the models are trained using these configurations on the full respective training datasets. After training is complete, forecasts are performed using the respective energy demand test datasets for one, four, and 24-hour horizons. Upon inverse normalization of the forecasting outcomes, we assess the results using MASE_{1d} as the primary forecasting performance evaluation metric, in conjunction with RMSE, expressed in kWh.

To increase to the robustness of the results, accounting for potential variations in model performance due to random factors such as initialization [208], this procedure is repeated five times and the evaluation results are averaged.

The results of the performance evaluation can be found in Tables 5.3, 5.4, and 5.5. It is important to note that each pair of scores in these tables represents a distinct model with a unique hyperparameter configuration.

1 hour	ACN Caltech		ACN JPL		Palo Alto		Boulder	
$MASE_{1d}$ || RMSE								
LinR	0.6464	4.6200	0.3815	7.0686	0.4931	8.0812	0.5544	3.5926
AdaBoost	0.5954	4.2810	0.3859	7.0581	0.4965	8.1245	0.6063	3.8582
XGBoost	**0.4218**	**3.1615**	**0.2231**	**4.2618**	**0.4431**	**7.2971**	**0.5202**	**3.3785**
Conv1D	0.5561	4.0497	0.2779	5.1558	0.4577	7.5288	0.5361	3.4778
LSTM	0.5131	3.6485	0.2812	5.2763	0.4447	7.3066	0.5450	3.5047
GRU	0.4947	3.5587	0.3249	6.0589	0.4711	7.7141	0.5632	3.6041
BiLSTM	0.5454	4.0810	0.3530	6.6476	0.4590	7.5231	0.5429	3.5095
ConvLSTM	0.4662	3.4823	0.2703	5.0173	0.4508	7.4771	0.5330	3.4578
CNN-LSTM	0.4671	3.4917	0.2463	4.4972	0.4545	7.4852	0.5436	3.5116
Stacked LSTM	0.4602	3.4270	0.3241	6.2678	0.4528	7.4436	0.5287	3.4444
Hybrid LSTM	0.4544	3.3858	0.2776	5.1449	0.4619	7.5799	0.5240	3.3882
PLCnet	0.5105	3.7869	0.2419	4.5810	0.4714	7.7097	0.5234	3.3978

Table 5.3: Predictive performance of baseline models, deep learning models and benchmark models in four-step ahead energy demand forecasting.

4 hours	ACN Caltech		ACN JPL		Palo Alto		Boulder	
$MASE_{1d}$ || RMSE								
LinR	0.8444	6.6491	0.6788	13.6694	0.7115	12.3064	0.7373	5.1260
AdaBoost	1.0181	7.9545	0.7695	15.3521	0.9061	15.3300	0.9486	6.4410
XGBoost	0.6995	5.4796	0.3732	7.8270	0.5904	10.4457	0.6993	4.8161
Conv1D	0.7940	6.1779	0.4216	8.6451	0.6261	10.9553	0.7198	5.0182
LSTM	0.6867	5.4269	0.5237	10.6675	0.5951	10.4220	0.7456	5.1729
GRU	0.7205	5.7122	0.5250	10.8929	0.6188	11.0598	0.6951	4.9439
BiLSTM	0.7698	6.1024	0.4375	9.2822	0.6028	10.5848	0.6924	4.9028
ConvLSTM	0.6417	5.1937	0.4137	9.1071	0.6078	10.8287	0.6867	4.8739
CNN-LSTM	**0.6383**	**4.9651**	**0.3656**	**7.5752**	**0.5788**	**10.3102**	**0.6743**	**4.7769**
Stacked LSTM	0.7964	6.1765	0.4384	9.1735	0.6041	10.5668	0.6933	4.8325
Hybrid LSTM	0.7291	5.7012	0.4043	8.5723	0.6030	10.7683	0.6865	4.9028
PLCnet	0.6758	5.2808	0.3880	8.2139	0.5968	10.4256	0.7004	4.9057

Table 5.4: Predictive performance of baseline models, deep learning models and benchmark models in 16-step ahead energy demand forecasting.

24 hours	ACN Caltech		ACN JPL		Palo Alto		Boulder	
$MASE_{1d}$ || RMSE								
LinR	1.1178	10.1418	0.7264	19.0502	0.8006	16.0363	0.8231	6.9130
AdaBoost	1.7458	15.3360	1.2643	21.1336	1.2643	21.1336	1.5452	10.1607
XGBoost	0.9816	8.6405	0.5759	14.5880	0.6785	14.0257	0.8592	6.9269
Conv1D	0.8352	8.4200	0.5773	15.4497	0.7025	14.4569	0.8032	7.1395
LSTM	0.8227	8.2782	0.6368	17.1560	0.7875	15.2892	0.8652	7.4264
GRU	0.8287	8.6426	0.6212	16.9940	0.7231	14.4180	0.8489	6.8215
BiLSTM	0.7830	7.6413	0.5006	13.3732	**0.6621**	**13.8017**	**0.7624**	**6.7249**
ConvLSTM	0.8039	8.5783	0.5777	14.4874	0.6970	14.3648	0.7896	6.8376
CNN-LSTM	**0.7419**	**7.5234**	**0.4823**	**13.0530**	0.6688	13.8775	0.7745	6.7160
Stacked LSTM	0.8258	8.4027	0.5246	14.4988	0.6840	14.2258	0.8115	7.2552
Hybrid LSTM	0.8385	9.1748	0.5654	15.2530	0.7004	14.5579	0.7852	6.9613
PLCnet	0.7622	7.9084	0.5540	14.9710	0.6852	14.2700	0.7811	6.7345

Table 5.5: Predictive performance of baseline models, deep learning models and benchmark models in 96-step ahead energy demand forecasting.

When analyzing the results at different forecasting horizons, we can observe that all models, except for AdaBoost, outperform the naïve forecast across all considered horizons. Another exception is the Linear Regression model, which demonstrates poor performance when forecasting ACN Caltech energy demand at a 24-hour horizon.

In general, the ensemble of models comprising multi-stream architectures and those that integrate recurrent, convolutional, and fully connected components demonstrates superior performance compared to the group of models solely composed of recurrent and dense or convolutional and dense layers. On average, across all datasets, the former group achieves a $MASE_{1d}$ score improvement of 5.19% for one-hour, 7.62% for four-hour, and 4.09% for 24-hour forecasts relative to the latter group. The CNN-LSTM architecture is the best performer on the ACN datasets for 16-hour and 24-hour horizons.

An exception to this observation is the BiLSTM model, which generates the most accurate overall forecasts at a 24-hour horizon on the city-wide time series.

A notable result is the overall performance of XGBoost. At a one-hour horizon, it exhibits an average improvement of 3.37% in terms of $MASE_{1d}$ compared to the second-best forecaster. At a 4-hour horizon, it still showcases an acceptable performance on all datasets. At a 24-hour horizon, XGBoost performs 32.29% worse than the best-performing model on the ACN Caltech series, but maintains acceptable predictive performance on the remaining datasets.

The overall superior performance of models that combine convolutional, fully connected, and recurrent components over the simpler deep learning models could be attributed to the synergistic effect resulting from integrating more components. By incorporating recurrent and convolutional layers, the models can use the strengths of each component type. The CNN serves as a trainable feature detector that extracts local hierarchical patterns, while the LSTM processes a sequence of high-level representations, which facilitates the learning of temporal structure. The convolutional component can further enhance the model's robustness to noise through its regularizing effect and the dimensionality reduction achieved by pooling layers can improve the model's generalization ability. The PLCnet model can be considered an extended version of this, where the extraction of local trends and learning of longer-term dependencies is initially decoupled. Although the Hybrid LSTM model does not include convolutional components (in most cases), its multi-stream setup might still augment its ability to optimally combine the output of its two predictors, where the LSTM component learns temporal dynamics and the fully connected input stream functions as a feature extractor.

The BiLSTM model appears to be a promising choice for forecasting at a 24-hour horizon, particularly for the city-wide time series, which generally exhibit less pronounced seasonal components, as discussed in Section 4.2.3. The model's bidirectional nature effectively captures forward and backward temporal dependencies. However, its performance may decrease on longer horizons due to limitations in extracting meaningful features from data with a more sparse representation in the time domain, like the ACN time series. In contrast, the CNN-LSTM model excels at learning temporal dynamics and handling data with sparse, complex patterns. This advantage is maintained even when the forecasting horizon is increased. On time series with overall less complex seasonal features, BiLSTM can fully leverage its ability to model long-term temporal dynamics.

The observed patterns in XGBoost's forecasting performance, relative to other models across different horizons and datasets, may stem from several factors. The ACN Caltech time series displays prominent pattern irregularities between the months of November 2018 and February 2019, as shown in Figure 4.4. A potential explanation for XGBoost's decreasing performance, particularly on this dataset, may be overfitting, a problem that can arise for the algorithm if the number of trees or their depth is not properly tuned, which might be the case here, as only a relatively small space was explored during hyperparameter optimization.

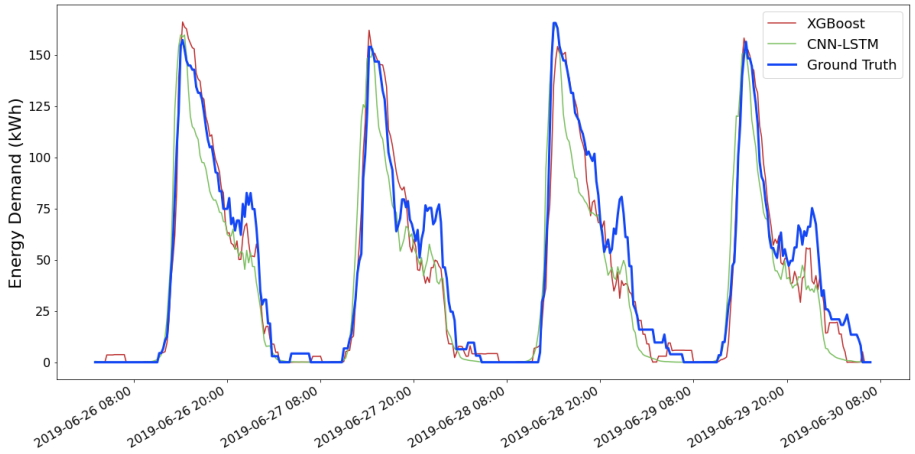


Figure 5.2: Energy demand forecasts for the ACN JPL time series. Models were trained for 4-step ahead forecasting.

When examining the two energy demand forecasts on the ACN JPL time series, as shown in Figures 5.2 and 5.3, another issue becomes apparent. As the models use a sliding window with stride length one, the depicted curves are a concatenation of the respective last predicted values in each forecast. In Figure 5.2, which displays forecasts performed by models trained for a 4-step horizon, it can be observed that the XGBoost model makes a closer overall prediction of the ground truth within the examined interval, compared to the CNN-LSTM forecaster. However, Figure 5.3 reveals a noticeable gap between XGBoost's predictions and the ground truth during zero-valued observation intervals. These plots are concatenations of the predicted last steps of the respective models trained for 96-step-ahead forecasting, using a sliding window with stride length one.

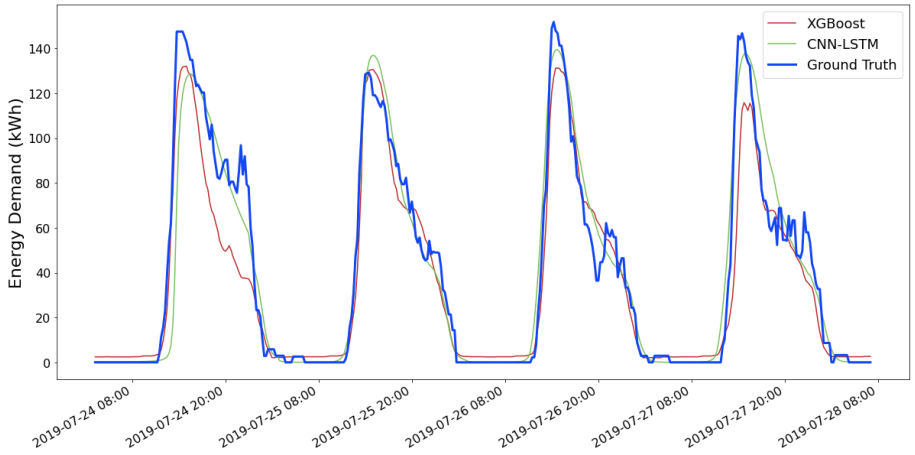


Figure 5.3: Energy demand forecasts for the ACN JPL time series. Models were trained for 96-step ahead forecasting.

The gap between the forecasted values and the ground truth could be attributable to error accumulation during recursive forecasting. A potential combination of overfitting and error accumulation in recursive forecasts could account for the rapid decline in XGBoost's performance compared to other models on the ACN Caltech series. As error accumulation is expected for all models not employing the MIMO strategy, the predictive performance of the benchmark models and Linear Regression at larger horizons might be distorted.

Overall, the performance of AdaBoost, as evaluated with the specified hyperparameter space, indicates that it is not a suitable benchmark model for EV energy demand forecasting. Potential reasons for this suboptimal performance include the reliance on shallow decision tree regressors as weak learners, which, even when utilizing lagged features and exogenous variables, may struggle to adequately capture the intricate relationships present in the given time series data. Also, the anticipated error accumulation during recursive forecasting might contribute to the subpar results showcased in Tables 5.3, 5.4 and 5.5.

5.2 Occupancy

5.2.1 Hyperparameters

Compared to the regression task, slightly less complex models tend to perform better. When averaged across datasets, the optimal hyperparameter configurations exhibit similar trends to those observed in the energy demand scenario. Specifically, the selected configurations display a combination of parameter count, lookback window size, and average dropout rate of 128.492, 9.7, and 0.23, respectively, when averaged across models and datasets.

Comparing the average model complexity of the best-performing configurations on the city-wide series to their counterparts on the ACN series reveals a trend that is reversed from that observed in the energy demand scenario, but less pronounced. Specifically, the best configurations on the city-wide series have an average parameter count that is only 4% larger than those performing best on the ACN series. Due to the heterogeneity of the underlying datasets with respect to factors such as average occupancy rates and the number of time series for each site, interpreting this result is difficult.

Considering the optimizer, 21 out of 32 selected model configurations use ADAM, while the rest use SGD. The overall relative training time of the benchmark models, compared to the deep learning models, is observed to be analogous to the regression scenario. Since the hyperparameter landscape exhibits high variability, we focus on the specific hyperparameter configurations of the architecture that ultimately demonstrates the best forecasting performance on the test time series, which is Hybrid LSTM. The respective hyperparameter configurations are displayed in Table 5.6.

2 hours	ACN Caltech	ACN JPL	Palo Alto	Boulder
Window Size p	16	4	8	8
Optimizer	ADAM	ADAM	ADAM	ADAM
Clip Norm	2.0	4.0	4.0	2.0
Batch Size	128	32	64	64
Epochs	32	64	256	64
Include Conv.	False	True	False	False
Stack Layers	2	2	2	3
Filters 1	–	16	–	–
Filters 2	–	16	–	–
Nodes Dense 1	64	8	32	16
Nodes Dense 2	16	16	4	128
Nodes Dense 3	4	32	32	8
Nodes Dense 4	8	–	8	32
Nodes Dense 5	8	–	128	4
Nodes Rec. 1	128	64	32	32
Nodes Rec. 2	16	32	128	64
Nodes Rec. 3	–	–	–	16
Dropout 1	0.25	0.03	0.47	0.22
Dropout 2	0.16	0.21	0.08	0.20
Weight Reg. 1	None	L_1	None	None
Weight Reg. 2	L_2	None	L_1	None
Weight Reg. 3	None	None		L_2
Weight Reg. 4	None	None	None	–
Weight Reg. 5	None	L_2	None	–

Table 5.6: Hyperparameter configurations of the overall best-performing occupancy forecasting models. The models were trained using an eight-step ahead forecasting horizon.

When examining Table 5.6, the models' tendency to perform better when stacking recurrent layers becomes apparent. In the case of Hybrid LSTM, ADAM appears to be the favorable optimizer. No clearly discernible patterns emerge regarding batch size, number of epochs, or regularization hyperparameters. The lower overall number of neurons in the fully connected layers of the ACN JPL forecaster can be attributed to the inclusion of two convolutional layers, which enable the fully connected hidden layers to operate on a condensed representation of the feature space.

5.2.2 Model Evaluation

Once again, models are trained using their respective favorable hyperparameter configurations on the full training datasets, and adjustments are made to the respective output layer sizes. This encompasses training each model on *all* datasets for each site, for each forecast horizon $h \in \{1, 4, 16\}$. After training is complete, one- four- and 16-step ahead forecasts are performed using the respective test dataset. We assess the results using F1-score as the primary forecasting performance evaluation metric, in conjunction with accuracy.

Due to the high overall number of datasets, we forego repeating this process several times and assume sufficient robustness of the results. This assumption is based on the fact that each model is trained and evaluated on at least 20 distinct time series to obtain a single score.

15 min	ACN Caltech		ACN JPL		Palo Alto		Boulder	
Accuracy || F1-Score								
LogR	0.9907	0.9890	0.9850	0.9823	0.9569	0.9510	0.9823	0.9795
AdaBoost	0.9901	0.9950	0.9823	0.9908	0.9549	0.9775	0.9771	0.9885
XGBoost	0.9906	0.9953	0.9852	0.9926	0.9568	0.9784	0.9822	0.9911
Conv1D	0.9890	0.9921	0.9832	0.9912	0.9436	0.9712	0.9798	0.9880
LSTM	0.9902	0.9951	0.9838	0.9919	0.9560	0.9774	0.9812	0.9906
GRU	0.9919	0.9960	0.9842	0.9915	0.9513	0.9749	0.9817	0.9911
BiLSTM	0.9912	0.9956	0.9837	0.9921	0.9451	0.9725	0.9744	0.9861
ConvLSTM	0.9920	0.9962	0.9847	0.9924	0.9522	0.9748	0.9730	0.9824
CNN-LSTM	0.9910	0.9929	0.9823	0.9912	0.9478	0.9756	0.9702	0.9810
Stacked LSTM	0.9916	0.9958	0.9845	0.9922	0.9569	0.9784	0.9820	0.9910
Hybrid LSTM	**0.9956**	**0.9998**	**0.9917**	**0.9964**	**0.9620**	**0.9822**	**0.9908**	**0.9942**

Table 5.7: Predictive performance of baseline models, deep learning models and benchmark models in one-step ahead charging station occupancy state forecasting.

1 hour	ACN Caltech		ACN JPL		Palo Alto		Boulder	
Accuracy || F1-Score								
LogR	0.9773	0.9722	0.9636	0.9555	0.9103	0.8913	0.9586	0.9493 1
AdaBoost	0.9768	0.9849	0.9530	0.9705	0.9020	0.9358	0.9557	0.9709
XGBoost	0.9790	0.9863	0.9635	0.9763	0.9028	0.9365	0.9584	0.9726
Conv1D	0.9635	0.9725	0.9600	0.9742	0.8926	0.9280	0.9351	0.9570
LSTM	0.9693	0.9802	0.9592	0.9735	0.8913	0.9298	0.9572	0.9715
GRU	0.9694	0.9804	0.9587	0.9727	0.9035	0.9359	0.9488	0.9635
BiLSTM	0.9759	0.9844	0.9612	0.9742	0.8893	0.9176	0.9576	0.9713
ConvLSTM	0.9817	0.9879	0.9526	0.9719	0.9076	0.9398	0.9522	0.9684
CNN-LSTM	0.9799	0.9869	0.9530	0.9707	0.8962	0.9319	0.9601	0.9746
Stacked LSTM	0.9759	0.9843	0.9591	0.9730	0.8981	0.9342	0.9583	0.9724
Hybrid LSTM	**0.9880**	**0.9910**	**0.9768**	**0.9826**	**0.9395**	**0.9544**	**0.9741**	**0.9804**

Table 5.8: Predictive performance of baseline models, deep learning models and benchmark models in 4 step-ahead charging station occupancy state forecasting.

4 hours	ACN Caltech		ACN JPL		Palo Alto		Boulder	
Accuracy || F1-Score								
LogR	0.9315	0.9020	0.8874	0.8447	0.7879	0.6904	0.9067	0.8481
AdaBoost	0.9220	0.9381	0.8718	0.9021	0.7891	0.8284	0.9022	0.9132
XGBoost	0.9153	0.9309	0.8974	0.9164	0.7980	0.8147	0.9125	0.9092
Conv1D	0.9270	0.9405	0.8813	0.9007	0.8172	0.8210	0.9120	0.9022
LSTM	0.9251	0.9372	0.8834	0.8996	0.8112	0.8169	0.9126	0.8820
GRU	0.9213	0.9389	0.9086	0.9205	0.8208	0.8253	0.9104	0.8966
BiLSTM	0.9363	0.9452	0.9022	0.9168	0.8016	0.8056	0.9100	0.9049
ConvLSTM	0.9185	0.9252	0.8803	0.8936	0.8046	0.8109	0.9150	0.9076
CNN-LSTM	0.9074	0.9175	0.9064	0.9140	0.7983	0.8061	0.9109	0.8981
Stacked LSTM	0.9278	0.9384	0.8978	0.9102	0.8134	0.8185	0.9115	0.9087
Hybrid LSTM	**0.9456**	**0.9536**	**0.9124**	**0.9248**	**0.8353**	**0.8366**	**0.9271**	**0.9199**

Table 5.9: Predictive performance of baseline models, deep learning models and benchmark models in 16-step ahead charging station occupancy state forecasting.

When examining the results presented in Tables 5.7, 5.8 and 5.9, it is crucial to note that the displayed scores represent averages computed by aggregating the performance scores of each model across all charging station occupancy time series generated from the datasets introduced in Section 4.2.

The analysis of these results reveals a consistent performance advantage for the Hybrid LSTM model in comparison to all other evaluated models across all datasets and examined forecasting horizons, as measured by both error evaluation metrics. While the performance of other models on the shorter forecasting horizon of 15 minutes ap-

pears balanced, the BiLSTM model demonstrates superior performance on the ACN datasets as the horizons increase. Meanwhile, AdaBoost is the second-best performer on the Palo Alto and Boulder time series for larger horizons.

AdaBoost's improved performance in the classification task can be ascribed to the inherently better ability of its underlying weak learners to capture simple binary relationships. In general, since predictions are constrained to one of two possible outcomes, we expect error accumulation during recursive multistep forecasting of binary time series to be less severe compared to multistep regression scenarios. In the latter case, small errors in early steps can accumulate and significantly impair the overall forecasting accuracy. This notion is reflected in the overall absence of deteriorating forecasting quality for the AdaBoost and XGBoost models at increasing forecasting horizons in the occupancy forecasting task.

The consistently high accuracy scores of the Logistic Regression model, along with relatively lower F1-Scores, indicate the model's limited capability to correctly predict the minority class, which suggests its limited suitability for occupancy forecasting.

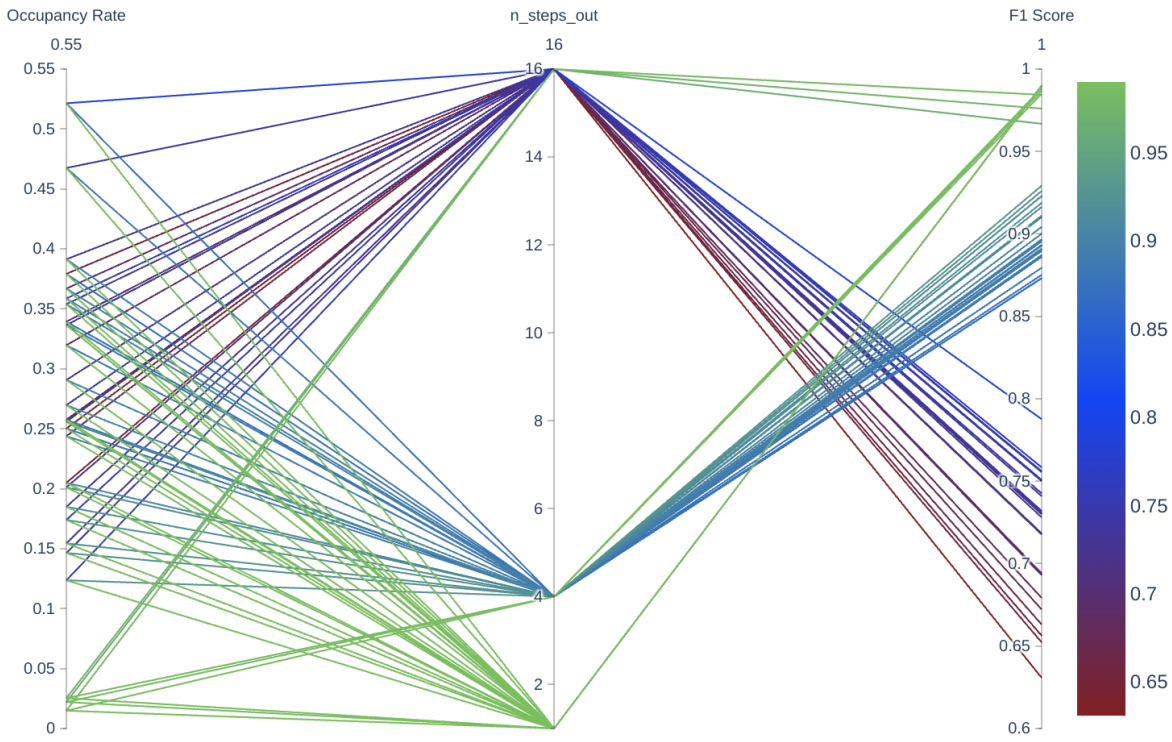


Figure 5.4: Parallel coordinates plot [209] illustrating the relationship between average occupancy rates of individual charging stations and predictive accuracy across three forecasting horizons, using the Palo Alto dataset.

Generally, predictive performance, as assessed by accuracy and F1-Score, tends to be high when forecasting occupancy for charging stations with a low average occupancy rate. This relationship is exemplified for the Palo Alto dataset in Figure 5.4.

The superior performance of the Hybrid LSTM model can be attributed to its method of combining outputs from different predictors. The model effectively enhances its predictive capabilities for potentially highly irregular time series, such as charging station occupancy state sequences, by learning to optimally combine the output of two independent predictors. One predictor is dedicated to learning (relatively) short-term temporal patterns, while the other focuses on long-term charging occupancy trends. Their outputs are integrated using multiple fully connected layers, allowing the overall model to learn from complementary aspects of the data.

In contrast, single-stream deep learning models directly process the combined input rather than learning from each source independently, which limits their ability to optimally combine local temporal patterns and long-term trends.

Overall, the outcomes of occupancy state forecasting support the conclusions drawn by the authors in [53]. The proposed Hybrid LSTM architecture demonstrates its ability to achieve superior performance on an extensive, heterogenous set of charging station occupancy state time series when compared to a range of alternative deep learning models, Logistic Regression, and AdaBoost. Additionally, it surpasses the well-regarded XGBoost in terms of performance.

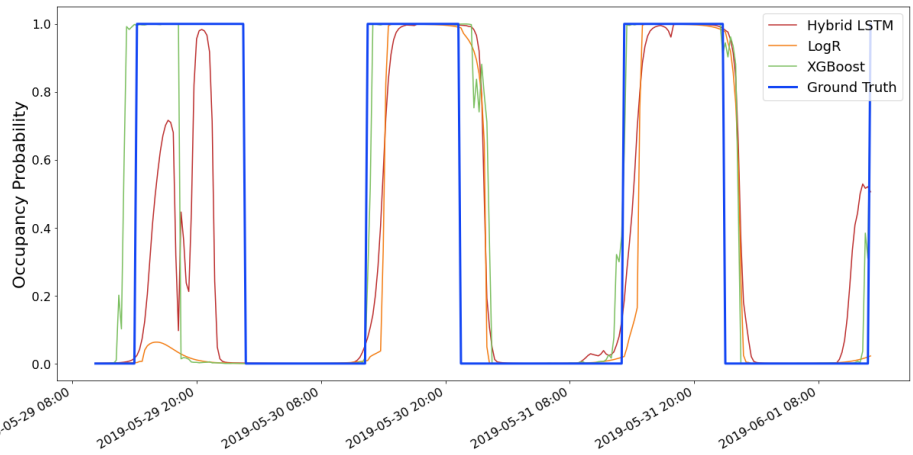


Figure 5.5: Occupancy probability forecasts for charging station ACN JPL 17. The models were trained for 16-step ahead forecasting. Logistic Regression shows a tendency to miss entire charging sessions, indicating its inability to accurately predict the minority class. XGBoost's predictions present a sharp and jagged appearance, while the predictions generated by Hybrid LSTM offer the most accurate overall fit to the ground truth.

Chapter 6

Conclusions

As in the previous sections, conclusions related to the two distinct forecasting tasks are addressed separately in their respective subsections.

6.1 Energy Demand

In conclusion, the evaluation of deep learning models for EV energy demand forecasting suggests that these architectures demonstrate their strengths particularly when two factors coexist: relatively long forecasting horizons and a high number of prominent seasonal components. On time series comprising a mixture of various frequencies due to more stochastic underlying user behavior, such as the Boulder time series, the examined architectures only exhibit a limited competitive advantage compared to Linear Regression, even at longer forecasting horizons. Given that almost no fine-adjustments were performed on Linear Regression, and it might potentially suffer from error accumulation, the overall advantage of employing complex deep learning models, which require significant time to implement and optimize, for energy demand forecasting on this type of time series can be deemed limited.

Among the examined deep learning models, the CNN-LSTM architecture is the best overall performer for the energy demand forecasting task. It shows strong predictive performance across all considered horizons and datasets. The model does particularly well at 24-hour forecasts on time series with large amplitude seasonal fluctuations and generalizes well to scenarios with more stochastic underlying user behavior. The BiLSTM model, with its bidirectional architecture, shows promise for 24-hour horizon forecasting, particularly on datasets with less pronounced seasonal components.

Overall, the group comprising multi-stream models, Hybrid LSTM and PLCnet, and the single-stream models that combine convolutional and recurrent layers, ConvLSTM

and CNN-LSTM, yields more promising results than the remaining models, with the exception of BiLSTM. We attribute this advantage to the synergistic effect of employing a combination of different component types. However, from a practical standpoint, it must be considered that models from this group also come with a relatively higher implementation complexity.

Despite these promising findings, it is crucial to consider the limitations of the experiments conducted on energy demand time series within this work, particularly regarding the overall performance of XGBoost. On one hand, no model outperforms XGBoost in energy demand forecasting with a one-hour horizon, where the boosting algorithm's advantage is especially pronounced on time series with a higher number of distinct seasonal components. On the other hand, its performance deteriorates significantly on a large horizon on one specific dataset, which might be attributable to a combination of overfitting and error accumulation in recursive forecasting.

In summary, determining whether the examined deep learning models are inherently better suited for energy demand time series forecasting than XGBoost remains a topic that should be addressed in future work. This includes exploring a larger hyperparameter space for the boosting algorithm and investigating more robust multi-step forecasting strategies.

There are several further limitations that need to be addressed concerning energy demand forecasting within this work. Only one feature set was used; assessing how different models perform with varying feature sets could be important in evaluating their suitability for the forecasting task. Hyperparameter optimization was carried out using the relatively basic random search approach; employing more advanced techniques like Bayesian search might yield better results but was beyond the scope of this work due to the number of evaluated models and time constraints. Access to a larger amount of computational resources could also improve the process of finding better hyperparameter configurations, especially considering the relatively low number of trials chosen for random search in this work. Lastly, it would be beneficial to compare deep learning models with other statistical benchmark methods commonly used for multivariate time series regression tasks, such as Vector Autoregression [210] or Multivariate Exponential Smoothing [211].

6.2 Occupancy

Unlike in the previous discussion on energy demand forecasting, the results of model evaluation for occupancy state forecasting of individual charging stations exhibit a clear preference for a specific deep learning-based architecture: The Hybrid LSTM model demonstrates the most favorable performance across all time series and forecasting horizons. Notably, one of our custom modifications, which can reduce overall model complexity by incorporating two convolutional layers instead of fully connected layers, appears to yield results comparable to the original architecture.

Regarding other model architectures, based on the results obtained within this work, no clearly discernible patterns emerge from the forecast evaluations. It is noteworthy that both benchmark models consistently perform on par with deep learning models across datasets and forecasting horizons, while Logistic Regression exhibits limited capabilities in accurately predicting occupancy states. Overall, we assume that in the occupancy forecasting task, the forecasters relying on the recursive multistep approach suffer less from error accumulation due to the binary nature of the problem.

We attribute the superior performance of the multi-stream Hybrid LSTM model to its ability to learn how to optimally combine the capabilities of its two independent predictors.

In general, the limitations concerning the occupancy prediction experiments are similar to the ones mentioned in the previous section. The problem of constrained computational resources is more severe within this task, as there are a total of 150 different datasets. Consequently, hyperparameter optimization could only be conducted using randomly sampled subsets of the full available data during each trial, which might limit the overall quality of the obtained hyperparameter configurations. Moreover, hyperparameters were only optimized for one forecasting horizon, and we assume that better overall predictive performance could be achieved by following a more granular approach.

Possible future work could involve evaluating modified versions of the Hybrid LSTM architecture, as it is easily extendable. Evaluation of the model's performance on other datasets could help in further confirming its superior applicability for the task of EV charging station occupancy state forecasting.

Bibliography

[1] EEA, "National emissions reported to the unfccc and to the eu greenhouse gas monitoring mechanism," 2020.

[2] B. Mathiesen, H. Lund, D. Connolly, H. Wenzel, P. Østergaard, B. Möller, S. Nielsen, I. Ridjan, P. Karnøe, K. Sperling, and F. Hvelplund, "Smart energy systems for coherent 100*Applied Energy*, vol. 145, pp. 139–154, 2015.

[3] B. Mathiesen, D. Connolly, H. Lund, M. Nielsen, E. Schaltz, H. Wenzel, N. Bentsen, C. Felby, P. Kaspersen, I. Skov, and K. Hansen, "Ceesa 100% renewable energy transport scenarios towards 2050," 06 2014.

[4] statista, "Projected ev sales in europe," 2023.

[5] Z. Yang, K. Li, and A. Foley, "Computational scheduling methods for integrating plug-in electric vehicles with power systems: A review," *Renewable and Sustainable Energy Reviews*, vol. 50, pp. 396–416, 11 2015.

[6] Bundesregierung, "Klimaschutzprogramm 2030 der bundesregierung zur umsetzung des klimaschutzplans 2050," 2020.

[7] E. Parliament and the Council, "Directive 2014/94/eu of the european parliament and of the council on the deployment of alternative fuels infrastructure. ausgefertigt am 2014-10-22, version vom 2017-11-18," 2017.

[8] M. Coffman, P. Bernstein, and S. Wee, "Electric vehicles revisited: a review of factors that affect adoption," *Transport Reviews*, vol. 37, pp. 1–15, 08 2016.

[9] Z. Rezvani, J. Jansson, and J. Bodin, "Advances in consumer electric vehicle adoption research: A review and research agenda," *Transportation Research Part D: Transport and Environment*, vol. 34, 01 2015.

[10] A. Ostermann, Y. Fabel, K. Ouan, and H. Koo, "Forecasting charging point occupancy using supervised learning algorithms," *Energies*, vol. 15, p. 3409, 05 2022.

[11] D. Pevec, J. Babic, A. Carvalho, Y. Ghiassi-Farrokhfal, W. Ketter, and V. Podobnik, "A survey-based assessment of how existing and potential electric vehicle owners perceive range anxiety," *Journal of Cleaner Production*, vol. 276, p. 122779, 07 2020.

[12] L. Noel, G. Zarazua de Rubens, B. Sovacool, and J. Kester, "Fear and loathing of electric vehicles: The reactionary rhetoric of range anxiety," *Energy Research Social Science*, vol. 48, pp. 96–107, 02 2019.

[13] D. Pevec, J. Babic, A. Carvalho, Y. Ghiassi-Farrokhfal, W. Ketter, and V. Podobnik, "Electric vehicle range anxiety: An obstacle for the personal transportation (r)evolution?," pp. 1–8, 06 2019.

[14] ACEA, "Fuel types of new cars," 2023.

[15] J. Zhu, Z. Yang, M. Mourshed, Y. Guo, Y. Zhou, Y. Chang, Y. Wei, and S. Feng, "Electric vehicle charging load forecasting: A comparative study of deep learning approaches," *Energies*, vol. 12, no. 14, 2019.

[16] M. Shepero, *Modeling and forecasting the load in the future electricity grid: Spatial electric vehicle load modeling and residential load forecasting*. PhD thesis, 10 2018.

[17] N. Daina, A. Sivakumar, and J. Polak, "Modelling electric vehicles use: a survey on the methods," *Renewable and Sustainable Energy Reviews*, vol. 68, Part 1, pp. 447–460, 02 2017.

[18] M. Coban and S. Tezcan, "Analysis of impact of electric vehicles on distribution grid using survey data," pp. 1–4, 10 2019.

[19] M. Motz, J. Huber, and C. Weinhardt, "Forecasting bev charging station occupancy at work places," 09 2020.

[20] A. Luo, J. Yuan, F. Liang, Q. Yang, and D. Mu, "Load forecasting of electric vehicle charging station based on edge computing," in *2020 IEEE 3rd International Conference on Computer and Communication Engineering Technology (CCET)*, pp. 34–38, 2020.

[21] N. B. Vanting, Z. Ma, and B. N. Jørgensen, "A scoping review of deep neural networks for electric load forecasting," *Energy Informatics*, vol. 4, no. 2, pp. 1–13, 2021.

[22] B. Box, G. Jenkins, G. Reinsel, and G. Ljung, *Time Series Analysis: Forecasting and Control*, vol. 68. 01 2016.

[23] G. Dorffner, "Neural networks for time series processing," *Neural Network World*, vol. 6, 08 1996.

[24] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 4, 09 2014.

[25] G. E. P. Box and D. R. Cox, "An analysis of transformations," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 26, no. 2, pp. pp. 211–252, 1964.

[26] F. Dama and C. Sinoquet, "Analysis and modeling to forecast in time series: a systematic review," *CoRR*, vol. abs/2104.00164, 2021.

[27] M. Tkáč and R. Verner, "Artificial neural networks in business: Two decades of research," *Applied Soft Computing*, vol. 38, 10 2015.

[28] J. C. B. Gamboa, "Deep learning for time-series analysis," *CoRR*, vol. abs/1701.01887, 2017.

[29] S. Sengupta, S. Basak, P. Saikia, S. Paul, V. Tsalavoutis, F. Atiah, R. Vadlamani, and R. II, "A review of deep learning with special emphasis on architectures, applications and recent trends," 02 2019.

[30] J. Zhu, Z. Yang, Y. Chang, Y. Guo, K. Zhu, and J. Zhang, "A novel lstm based deep learning approach for multi-time scale electric vehicles charging load prediction," in *2019 IEEE Innovative Smart Grid Technologies-Asia (ISGT Asia)*, pp. 3531–3536, IEEE, 2019.

[31] C. Ye, "Electric load data characterizing and forecasting based on trend index and auto-encoders," *The Journal of Engineering*, vol. 2018, 09 2018.

[32] C. Tong, J. Li, C. Lang, F. Kong, J. Niu, and J. Rodrigues, "An efficient deep model for day-ahead electricity load forecasting with stacked denoising auto-encoders," *Journal of Parallel and Distributed Computing*, vol. 117, 06 2017.

[33] S. Ryu, J. Noh, and H. Kim, "Deep neural network based demand side short term load forecasting," in *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pp. 308–313, 2016.

[34] D. L. Marino, K. Amarasinghe, and M. Manic, "Building energy load forecasting using deep neural networks," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, pp. 7046–7051, 2016.

[35] A. Lahouar and J. Ben Hadj Slama, "Random forests model for one day ahead load forecasting," 03 2015.

[36] G. Dudek, "A comprehensive study of random forest for short-term load forecasting," *Energies*, vol. 15, p. 7547, 10 2022.

[37] L. W. Chong, D. Rengasamy, Y. W. Wong, and R. K. Rajkumar, "Load prediction using support vector regression," in *TENCON 2017 - 2017 IEEE Region 10 Conference*, pp. 1069–1074, 2017.

[38] B. E. Türkay and D. Demren, "Electrical load forecasting using support vector machines," in *2011 7th International Conference on Electrical and Electronics Engineering (ELECO)*, pp. I–49–I–53, 2011.

[39] X. Lv, X. Cheng, YanShuang, and Y.-m. Tang, "Short-term power load forecasting based on balanced knn," *IOP Conference Series: Materials Science and Engineering*, vol. 322, p. 072058, 03 2018.

[40] M. Atanasovski, M. Kostov, B. Arapinoski, and M. Spirovski, "K-nearest neighbor regression for forecasting electricity demand," pp. 110–113, 09 2020.

[41] W. Ling, P. Zhang, Q. Li, D. Mo, C. Luo, Y. Li, and Z. Liang, "Adaboost-based power system load forecasting," *Journal of Physics: Conference Series*, vol. 2005, p. 012190, 08 2021.

[42] W. Lin, B. Zhang, H. Li, and R. Lu, "Short-term load forecasting based on eemd-adaboost-bp," *Systems Science & Control Engineering*, vol. 10, no. 1, pp. 846–853, 2022.

[43] X. Liao, N. Cao, M. Li, and X. Kang, "Research on short-term load forecasting using xgboost based on similar days," in *2019 International Conference on Intelligent Transportation, Big Data Smart City (ICITBS)*, pp. 675–678, 2019.

[44] R. Abbasi, N. Javaid, M. Ghuman, Z. Khan, S. Rehman, and A. Ullah, *Short Term Load Forecasting Using XGBoost*, pp. 1120–1131. 03 2019.

[45] H. Shi, M. Xu, Q. Ma, C. Zhang, R. Li, and F. Li, "A whole system assessment of novel deep learning approach on short-term load forecasting," *Energy Procedia*, vol. 142, pp. 2791–2796, 2017. Proceedings of the 9th International Conference on Applied Energy.

[46] C. Tian, J. Ma, C. Zhang, and P. Zhan, "A deep neural network model for short-term load forecast based on long short-term memory network and convolutional neural network," *Energies*, vol. 11, p. 3493, 12 2018.

[47] W. He, "Load forecasting via deep neural networks," *Procedia Computer Science*, vol. 122, pp. 308–314, 01 2017.

[48] B. Farsi, M. Amayri, N. Bouguila, and U. Eicker, "On short-term load forecasting using machine learning techniques and a novel parallel deep lstm-cnn approach," *IEEE Access*, vol. PP, pp. 1–1, 02 2021.

[49] X. Zhao, Q. Li, W. Xue, Y. Zhao, H. Zhao, and S. Guo, "Research on ultra-short-term load forecasting based on real-time electricity price and window-based xgboost model," *Energies*, vol. 15, p. 7367, 10 2022.

[50] D.-J. Bae, B.-S. Kwon, and K.-B. Song, "Xgboost-based day-ahead load forecasting algorithm considering behind-the-meter solar pv generation," *Energies*, vol. 15, p. 128, 12 2021.

[51] X. Zhang, "Short-term load forecasting for electric bus charging stations based on fuzzy clustering and least squares support vector machine optimized by wolf pack algorithm," *Energies*, vol. 11, p. 1449, 06 2018.

[52] C. Hecht, J. Figgener, and D. U. Sauer, "Predicting electric vehicle charging station availability using ensemble machine learning," *Energies*, vol. 14, no. 23, 2021.

[53] T.-y. Ma and S. Faye, "Multistep electric vehicle charging station occupancy prediction using hybrid lstm neural networks," *Energy*, vol. 244, p. 123217, 01 2022.

[54] F. Soldan, E. Bionda, G. Mauri, and S. Celaschi, "Short-term forecast of ev charging stations occupancy probability using big data streaming analysis," 04 2021.

[55] G. Gruosso, A. Mion, and G. Gajani, "Forecasting of electrical vehicle impact on infrastructure: Markov chains model of charging stations occupation," *eTransportation*, vol. 6, p. 100083, 09 2020.

[56] B. Mouaad, M. Farag, K. Benabdelaziz, T. Kousksou, and M. Zazi, "A deep learning approach for electric vehicle charging duration prediction at public charging stations: The case of morocco," vol. 43, p. 01024, 02 2022.

[57] M. Wolf, "Mathematical foundations of supervised learning," 2020.

[58] D. Nielsen, "Tree boosting with xgboost - why does xgboost win "every" machine learning competition?," 2016.

[59] A. Tewari and P. L. Bartlett, "Chapter 14 - learning theory," in *Academic Press Library in Signal Processing: Volume 1* (P. S. Diniz, J. A. Suykens, R. Chellappa, and S. Theodoridis, eds.), vol. 1 of *Academic Press Library in Signal Processing*, pp. 775–816, Elsevier, 2014.

[60] A. Rakotomamonjy, X. Mary, and S. Canu, "Non-parametric regression with wavelet kernels," *Applied Stochastic Models in Business and Industry*, vol. 21, pp. 153 – 163, 03 2005.

[61] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. `http://www.deeplearningbook.org`.

[62] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: a review," *Data Mining and Knowledge Discovery*, vol. 33, 07 2019.

[63] G. Chevillon, "Direct multi-step estimation and forecasting," *SSRN Electronic Journal*, 01 2006.

[64] F. Takens, "Detecting strange attractors in turbulence," *Dynamical Systems and Turbulance serial Lecture notes in Mathematics*, vol. 898, 01 1981.

[65] J. Brownlee, *Introduction to Time Series Forecasting With Python: How to Prepare Data and Develop Models to Predict the Future*. Machine Learning Mastery, 2017.

[66] R. Adhikari and R. Agrawal, *An Introductory Study on Time series Modeling and Forecasting*. 01 2013.

[67] C. Chatfield, W. Fuller, W. Enders, D. Cox, D. Hinkley, and O. Barndoff-Nielson, "The analysis of time series: An introduction," *Biometrics*, vol. 52, p. 1162, 09 1996.

[68] B. Bogacka, "Mth6139 time series," 2008.

[69] P. Diggle, "Time series: A biostatistical introduction," *Journal of the American Statistical Association*, vol. 87, 03 1992.

[70] H. Musbah, M. E. El-Hawary, and H. H. H. Aly, "Identifying seasonality in time series by applying fast fourier transform," *2019 IEEE Electrical Power and Energy Conference (EPEC)*, pp. 1–4, 2019.

[71] S. Subba Rao, *A Course in Time Series Analysis*. Wiley, 2005.

[72] S. Ben Taieb, G. Bontempi, A. Atiya, and A. Sorjamaa, "A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition," *Expert Systems with Applications*, vol. 39, 08 2011.

[73] A. Weigend and N. Gershenfeld, *Time Series Prediction: Forecasting the Future and Understanding the Past*. Addison-Wesley, 1993.

[74] G. Bontempi, "Long term time series prediction with multi-input multi-output local learning," 2008.

[75] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, vol. 5, pp. 157–66, 02 1994.

[76] R. Pascanu, G. Montufar, and Y. Bengio, "On the number of response regions of deep feed forward networks with piece-wise linear activations," 12 2013.

[77] N. Roux and Y. Bengio, "Deep belief networks are compact universal approximators," *Neural computation*, vol. 22, 03 2010.

[78] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approxmations neural networks 2 no 5, 359-366," *Neural Networks*, vol. 2, pp. 359–366, 12 1989.

[79] R. K. Wittmann, "Photorealistic simulation of photonic-integrated circuit designs using digital image processing techniques and neural networks," 2022.

[80] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[81] I. Sutskever, "Training recurrent neural networks," *University of Toronto, Toronto, Ont., Canada*, 2013.

[82] J. Schill, "Scaling up the equation-encoder - handling high data volume through the efficient use of trainable parameters," 2020.

[83] R. Rojas, *Neural networks: a systematic introduction*. Springer, 1996.

[84] M. Sangiorgio, F. Dercole, and G. Guariso, *Neural Approaches for Time Series Forecasting*, pp. 43–57. 12 2021.

[85] G. Garrigos and R. M. Gower, "Handbook of convergence theorems for (stochastic) gradient methods," 2023.

[86] R. Tibshirani, "Convex optimization," 2013.

[87] S. Ruder, "An overview of gradient descent optimization algorithms," 09 2016.

[88] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, "Robust stochastic approximation approach to stochastic programming," *Society for Industrial and Applied Mathematics*, vol. 19, pp. 1574–1609, 01 2009.

[89] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, 12 2010.

[90] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[91] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[92] Z. Zhang, "Improved adam optimizer for deep neural networks," pp. 1–2, 06 2018.

[93] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," 2019. cite arxiv:1910.05446.

[94] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278 – 2324, 12 1998.

[95] B. Lim and S. Zohren, "Time-series forecasting with deep learning: a survey," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 379, p. 20200209, feb 2021.

[96] A. Gasparin, S. Lukovic, and C. Alippi, "Deep learning for time series forecasting: The electric load case," *CAAI Transactions on Intelligence Technology*, vol. 7, 09 2021.

[97] I. Koprinska, D. Wu, and Z. Wang, "Convolutional neural networks for energy time series forecasting," in *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2018.

[98] L. Li, K. Ota, and M. Dong, "Everything is image: Cnn-based short-term electrical load forecasting for smart grid," in *2017 14th International Symposium on*

Pervasive Systems, Algorithms and Networks 2017 11th International Conference on Frontier of Computer Science and Technology 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC), pp. 344–351, 2017.

[99] E. Lewinson, *Python For Finance Cookbook*. Birmingham, UK: Packt Publishing, 1 ed., 2020.

[100] R. Srinivasamurthy, "Understanding 1d convolutional neural networks using multiclass time-varying signals," 2018.

[101] A. Zhang, Z. Lipton, M. Li, and A. Smola, *Dive into Deep Learning*. 06 2021.

[102] D. Krompaß, "Convolutional neural networks," 2018.

[103] S. Damelin and W. Miller, "The mathematics of signal processing," *The Mathematics of Signal Processing*, 01 2011.

[104] K. Benidis, S. S. Rangapuram, V. Flunkert, Y. B. Wang, D. Robinson, C. Turkmen, J. Gasthaus, M. Bohlke-Schneider, D. Salinas, L. Stella, F.-X. Aubet, L. Callot, and T. Januschowski, "Deep learning for time series forecasting: Tutorial and literature survey," *ACM Computing Surveys*, 2021.

[105] R. F. L. Neves, "An overview of deep learning strategies for time series prediction," 2018.

[106] K. O'Shea and R. Nash, "An introduction to convolutional neural networks.," *CoRR*, vol. abs/1511.08458, 2015.

[107] S. Rosset and J. Friedman, "Topics in regularization and boosting /," 02 2023.

[108] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[109] H. Kumar, "Dropout: Prevent overfitting," 2018.

[110] V. Cerqueira, L. Torgo, J. Smailovic, and I. Mozetic, "A comparative study of performance estimation methods for time series forecasting," pp. 529–538, 10 2017.

[111] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[112] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling.," *CoRR*, vol. abs/1803.01271, 2018.

[113] F. M. Bianchi, E. Maiorino, M. Kampffmeyer, A. Rizzi, and R. Jenssen, *Recurrent Neural Networks for Short-Term Load Forecasting: An Overview and Comparative Analysis*. 01 2017.

[114] P. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[115] C. C. Aggarwal, *Neural Networks and Deep Learning*. Cham: Springer, 2018.

[116] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," *30th International Conference on Machine Learning, ICML 2013*, 11 2012.

[117] L. Livi, F. M. Bianchi, and C. Alippi, "Determination of the edge of criticality in echo state networks through fisher information maximization," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, pp. 1–12, 01 2017.

[118] R. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, 09 1998.

[119] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[120] Y. Wu, M. Schuster, Z. Chen, Q. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, u. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," 09 2016.

[121] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pp. 338–342, 01 2014.

[122] H. Mayer, F. Gomez, D. Wierstra, I. Nagy, A. Knoll, and J. Schmidhuber, "A system for robotic heart surgery that learns to tie knots using recurrent neural networks.," *Advanced Robotics*, vol. 22, pp. 1521–1537, 01 2008.

[123] O. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research*, vol. 39, p. 027836491988744, 11 2019.

[124] J. Rodriguez, "The science behind openai five that just produced one of the greatest breakthrough in the history of ai," July 2018.

[125] K. Arulkumaran, A. Cully, and J. Togelius, "Alphastar: an evolutionary computation perspective," pp. 314–315, 07 2019.

[126] G. Van Kriekinge, C. De Cauwer, N. Sapountzoglou, T. Coosemans, and M. Messagie, "Day-ahead forecast of electric vehicle charging demand with deep neural networks," *World Electric Vehicle Journal*, vol. 12, no. 4, 2021.

[127] B. P. Orozco, "Recurrent neural networks for time series prediction," 2020.

[128] K. Cho, B. Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 09 2014.

[129] B. C. Mateus, M. Mendes, J. T. Farinha, R. Assis, and A. M. Cardoso, "Comparing lstm and gru models to predict the condition of a pulp paper press," *Energies*, vol. 14, no. 21, 2021.

[130] S. Nosouhian, F. Nosouhian, and A. Khoshouei, "A review of recurrent neural network architecture for sequence learning: Comparison between lstm and gru," 07 2021.

[131] P. T. Yamak, L. Yujian, and P. K. Gadosey, "A comparison between arima, lstm, and gru for time series forecasting," in *International Conference on Advances in Computing and Artificial Intelligence*, 2019.

[132] J. Zhu, Z. Yang, Y. Guo, J. Zhang, and H. Yang, "Short-term load forecasting for electric vehicle charging stations based on deep learning approaches," *Applied Sciences*, vol. 9, no. 9, 2019.

[133] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," 12 2013.

[134] A. Graves, "Generating sequences with recurrent neural networks," 08 2013.

[135] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," 02 2015.

[136] J. Chung, S. Ahn, and Y. Bengio, "Hierarchical multiscale recurrent neural networks," 09 2016.

[137] M. Hermans and B. Schrauwen, "Training and analysing deep recurrent neural networks," *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, pp. 190–198, 01 2013.

[138] R. Chandra, S. Goyal, and R. Gupta, "Evaluation of deep learning models for multi-step ahead time series prediction," *IEEE Access*, vol. PP, pp. 1–1, 05 2021.

[139] F. Xiao, "Time series forecasting with stacked long short-term memory networks," 11 2020.

[140] C. Cai, Y. Tao, T. Zhu, and Z. Deng, "Short-term load forecasting based on deep learning bidirectional lstm neural network," *Applied Sciences*, vol. 11, no. 17, 2021.

[141] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[142] I. Ihianle, A. Nwajana, S. Ebenuwa, R. Otuka, K. Owa, and M. Orisatoki, "A deep learning approach for human activities recognition from multimodal sensing devices," *IEEE Access*, vol. 8, pp. 179028–179038, 10 2020.

[143] S. Siami Namini, N. Tavakoli, and A. Siami Namin, "A comparative analysis of forecasting financial time series using arima, lstm, and bilstm," 11 2019.

[144] M. Yang and J. Wang, "Adaptability of financial time series prediction based on bilstm," *Procedia Computer Science*, vol. 199, pp. 18–25, 2022. The 8th International Conference on Information Technology and Quantitative Management (ITQM 2020 2021): Developing Global Digital Economy after COVID-19.

[145] Y. Shi and X. Xu, "Deep federated adaptation: An adaptative residential load forecasting approach with federated learning," *Sensors*, vol. 22, no. 9, 2022.

[146] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4580–4584, 2015.

[147] W. Lu, J. Li, Y. Li, A. Sun, and J. Wang, "A cnn-lstm-based model to forecast stock prices," *Complexity*, vol. 2020, pp. 1–10, 11 2020.

[148] I. Livieris, E. Pintelas, and P. Pintelas, "A cnn-lstm model for gold price time series forecasting," *Neural Computing and Applications*, vol. 32, 12 2020.

[149] L. Gong, Y. Chao, X. Huang, and J. Chen, "Application of cnn-lstm based hybrid neural network in power load forecasting," 06 2022.

[150] T. Y. Kim and S.-B. Cho, "Predicting residential energy consumption using cnn-lstm neural networks," *Energy*, 2019.

[151] H. Dong, J. Zhu, S. Li, T. Luo, H. Li, and Y. Huang, "Ultra-short- term load forecasting based on convolutional-lstm hybrid networks," in *2022 IEEE 31st International Symposium on Industrial Electronics (ISIE)*, pp. 142–148, 2022.

[152] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-k. Wong, and W.-c. Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, (Cambridge, MA, USA), p. 802–810, MIT Press, 2015.

[153] P. Zhang, L. Chen, Z. Li, J. Xing, X. Xing, and Y. Zhihui, "Automatic extraction of water and shadow from sar images based on a multi-resolution dense encoder and decoder network," *Sensors*, vol. 19, p. 3576, 08 2019.

[154] L. Li, C. Meinrenken, V. Modi, and P. Culligan, "Short-term apartment-level load forecasting using a modified neural network with selected auto-regressive features," *Applied Energy*, vol. 287, p. 116509, 04 2021.

[155] M. Schmitt, "Deep learning vs. gradient boosting: Benchmarking state-of-the-art machine learning algorithms for credit scoring," 2022.

[156] U. Pritzsche, "Benchmarking of classical and machine-learning algorithms (with special emphasis on bagging and boosting approaches) for time series forecasting," 2015.

[157] A. Jović, K. Brkić, and N. Bogunović, "Decision tree ensembles in biomedical time-series classification," in *Pattern Recognition* (A. Pinz, T. Pock, H. Bischof, and F. Leberl, eds.), (Berlin, Heidelberg), pp. 408–417, Springer Berlin Heidelberg, 2012.

[158] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *International Conference on Machine Learning*, pp. 148–156, 1996.

[159] Y. Freund and R. E. Schapire, "A Decision Theoretic Generalization of On-Line Learning and an Application to Boosting," in *Second European Conference on Computational Learning Theory (EuroCOLT-95)* (P. M. B. Vitányi, ed.), pp. 23–37, 1995.

[160] Y. Freund, "Boosting a weak learning algorithm by majority," *Information and Computation*, vol. 121, no. 2, pp. 256–285, 1995.

[161] J. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, vol. 29, pp. 1189–1232, 10 2001.

[162] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[163] H. Drucker, "Improving regressors using boosting techniques," in *International Conference on Machine Learning*, 1997.

[164] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer Series in Statistics, New York, NY, USA: Springer New York Inc., 2001.

[165] F. Sigrist, "Gradient and newton boosting for classification and regression," 2018.

[166] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system.," in *KDD* (B. Krishnapuram, M. Shah, A. J. Smola, C. Aggarwal, D. Shen, and R. Rastogi, eds.), pp. 785–794, ACM, 2016.

[167] W. Richert and L. P. Coelho, *Building Machine Learning Systems with Python*. Packt Publishing Ltd, 2013.

[168] B. LM, J. Friedman, R. Olshen, and C. Stone, *CART: Classification and Regression Trees*. 01 1983.

[169] J. R. Quinlan, "Learning with continuous classes," 1992.

[170] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is np-complete.," *Inf. Process. Lett.*, vol. 5, no. 1, pp. 15–17, 1976.

[171] B. D. Ripley, *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[172] C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn, "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC Bioinformatics*, vol. 8, no. 25, 2007.

[173] A. Ferrario and R. Hämmerli, "On boosting: Theory and applications," *Machine Learning eJournal*, 2019.

[174] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *NIPS*, 2017.

[175] S. Athiyarath, M. Paul, and S. Krishnaswamy, "A comparative study and analysis of time series forecasting techniques," *SN Computer Science*, vol. 1, 05 2020.

[176] L. Wasserman, *All of statistics : a concise course in statistical inference*. New York: Springer, 2010.

[177] P. MacCullagh and J. A. Nelder, *Generalized linear models*, vol. 37. CRC press, 1989.

[178] D. Tran, P. Toulis, and E. M. Airoldi, "Stochastic gradient descent methods for estimation with large data sets," *arXiv: Computation*, 2015.

[179] M. Zahid, F. Ahmed, N. Javaid, R. Abbasi, H. Kazmi, A. Javaid, M. Bilal, M. Akbar, and M. Ilahi, "Electricity price and load forecasting using enhanced convolutional neural network and enhanced support vector regression in smart grids," *Electronics*, vol. 8, p. 122, 01 2019.

[180] S. Makridakis, "Evaluating accuracy (or error) measures," 1995.

[181] R. Hyndman, "Another look at forecast accuracy metrics for intermittent demand," *Foresight: The International Journal of Applied Forecasting*, vol. 4, pp. 43–46, 01 2006.

[182] M. Fatourechi, R. K. Ward, S. G. Mason, J. Huggins, A. Schlögl, and G. E. Birch, "Comparison of evaluation metrics in classification applications with imbalanced datasets," in *2008 Seventh International Conference on Machine Learning and Applications*, pp. 777–782, 2008.

[183] H. Borchani, G. Varando, C. Bielza, and P. Larrañaga, "A survey on multi-output regression," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 5, 2015.

[184] Y. Li, Y. Huang, and M. Zhang, "Short-term load forecasting for electric vehicle charging station based on niche immunity lion algorithm and convolutional neural network," *Energies*, vol. 11, p. 1253, 05 2018.

[185] C. M. Castilho, "Time series forecasting with exogenous factors: Statistical vs machine learning approaches," 2020.

[186] S. Makridakis, R. J. Hyndman, and F. Petropoulos, "Forecasting in social settings: The state of the art," *International Journal of Forecasting*, vol. 36, pp. 15–28, 2020.

[187] I. Koprinska, M. Rana, and V. Agelidis, "Correlation and instance based feature selection for electricity load forecasting," *Knowledge-Based Systems*, vol. 82, 02 2015.

[188] M. H. Amini, A. Kargarian, and O. Karabasoglu, "Arima-based decoupled time series forecasting of electric vehicle charging demand for stochastic power system operation," *Electric Power Systems Research*, vol. 140, no. C, pp. 378–390, 2016.

[189] Y. Lu, Y. Li, D. Xie, E. Wei, X. Bao, H. Chen, and X. Zhong, "The application of improved random forest algorithm on the prediction of electric vehicle charging load," *Energies*, vol. 11, no. 11, 2018.

[190] L. Buzna, P. De Falco, S. Khormali, D. Proto, and M. Straka, "Electric vehicle load forecasting: A comparison between time series and machine learning approaches," pp. 1–5, 05 2019.

[191] M. Majidpour, C. Qiu, P. Chu, H. Pota, and R. Gadh, "Forecasting the ev charging load based on customer profile or station measurement?," *Applied Energy*, vol. 163, pp. 134–141, 02 2016.

[192] ACN, "Acn-data: Analysis and applications of an open ev charging dataset," 2023.

[193] C. of Boulder Open Data, "Boulder electric vehicle charging station energy consumption," 2023.

[194] C. of Palo Alto, "Electric vehicle charging station usage (july 2011 - dec 2020)," 2023.

[195] N. Kraftwerke, "Was ist der intraday-handel," 2020.

[196] A. Mansour-Saatloo, A. Moradzadeh, B. Mohammadi-Ivatloo, A. Ahmadian, and A. Elkamel, "Machine learning based pevs load extraction and analysis," *Electronics*, vol. 9, no. 7, 2020.

[197] A. Bui, P. Slowik, and N. Lutsey, "Evaluating electric vehicle market growth across u.s. cities," 09 2021.

[198] T. Mahajan, "An experimental assessment of treatments for cyclical data," 2021.

[199] L. Smith, "Cyclical learning rates for training neural networks," 10 2016.

[200] I. London, "Encoding cyclical continuous features - 24-hour time," 2016.

[201] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," 06 2017.

[202] Y. Lecun, L. Bottou, G. Orr, and K.-R. Müller, "Efficient backprop," 08 2000.

[203] A. Azadeh and S. Ghaderi, "Forecasting electrical consumption by integration of neural network, time series and anova," *Applied Mathematics and Computation*, vol. 186, pp. 1753–1761, 03 2007.

[204] K. Lakhwani, *Machine Learning*. 07 2021.

[205] F. Hutter, J. Lücke, and L. Schmidt-Thieme, "Beyond manual tuning of hyper-parameters," *KI - Künstliche Intelligenz*, vol. 29, 07 2015.

[206] J. Moolayil, *Learn Keras for Deep Neural Networks: A Fast-Track Approach to Modern Deep Learning with Python*. 01 2019.

[207] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[208] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms," *Neural Computation*, vol. 10, no. 7, pp. 1895–1923, 1998.

[209] P. T. Inc., "Collaborative data science," 2015.

[210] C. A. Sims, "Macroeconomics and reality," *Econometrica*, vol. 48, no. 1, pp. 1–48, 1980.

[211] A. C. Harvey, "Analysis and generalisation of a multivariate exponential smoothing model," *Management Science*, vol. 32, no. 3, pp. 374–380, 1986.

[212] M. Q. Raza and A. Khosravi, "A review on artificial intelligence based load demand forecasting techniques for smart grid and buildings," *Renewable and Sustainable Energy Reviews*, vol. 50, pp. 1352–1372, 2015.

[213] A. Dubey and S. Santoso, "Electric vehicle charging on residential distribution systems: Impacts and mitigations," *IEEE Access*, vol. 3, pp. 1871–1893, 2015.

[214] M. A. Hammad, B. Jereb, B. Rosi, and D. Dragan, "Methods and models for electric load forecasting: A comprehensive review," *Logistics, Supply Chain, Sustainability and Global Challenges*, vol. 11, no. 1, pp. 51–76, 2020.

[215] F. Liao, E. Molin, and B. Wee, "Consumer preferences for electric vehicles: a literature review," *Transport Reviews*, vol. 37, pp. 1–24, 09 2016.

[216] W. Sierzchula, S. Bakker, K. Maat, and B. Wee, "The influence of financial incentives and other socio-economic factors on electric vehicle adoption," *Energy Policy*, vol. 68, p. 183–194, 05 2014.

[217] Flurescu, *Probability and Stochastic Processes*. 2014.

[218] H. Cheng, P.-N. Tan, J. Gao, and J. Scripps, "Multistep-ahead time series prediction," 04 2006.

[219] A.-L. Cauchy, "Méthode générale pour la résolution des systèmes d'équations simultanées," *C. R. Acad. Sci. Paris*, vol. 25, pp. 536–538, 01 1847.

[220] J. Makin, "Backpropagation," 2006.

[221] M. Valueva, N. Nagornov, P. Lyakhov, G. Valuev, and N. Chervyakov, "Application of the residue number system to reduce hardware costs of the convolutional neural network implementation," *Mathematics and Computers in Simulation*, vol. 177, 05 2020.

[222] O. Abiodun, A. Jantan, O. Omolara, K. Dada, N. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, p. e00938, 11 2018.

[223] A. Tealab, "Time series forecasting using artificial neural networks methodologies: A systematic review," *Future Computing and Informatics Journal*, vol. 3, 11 2018.

[224] F. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, pp. 2451–71, 10 2000.

[225] F. Tajdini and R. Mehri, "Deep learning in healthcare," 11 2022.

[226] S. Ruder, "An overview of gradient descent optimization algorithms," 09 2016.

[227] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," *NIPS*, vol. 27, 06 2014.

[228] M. Shafiq and Z. Gu, "Deep residual learning for image recognition: A survey," *Applied Sciences*, 09 2022.

[229] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. Dauphin, "Convolutional sequence to sequence learning," 05 2017.

[230] A. oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 09 2016.

[231] A. KARPATHY, "The unreasonable effectiveness of recurrent neural networks," 2015.

[232] Y. Lu, Y. Li, D. Xie, E. Wei, X. Bao, H. Chen, and X. Zhong, "The application of improved random forest algorithm on the prediction of electric vehicle charging load," *Energies*, vol. 11, p. 3207, 11 2018.

[233] D. Solomatine and D. Shrestha, "Adaboost.rt: A boosting algorithm for regression problems," vol. 2, pp. 1163 – 1168 vol.2, 08 2004.

[234] S. Subbiah and J. Chinnappan, "A review of short term load forecasting using deep learning," *International Journal on Emerging Technologies*, vol. 11, pp. 378–384, 01 2020.

[235] A. Almalaq and G. Edwards, "A review of deep learning methods applied on load forecasting," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 511–516, 2017.

[236] S. Shrivastava, "Cross validation in time series," 2020.

[237] W. Fuller, *Introduction to statistical time series*. A Wiley publication in applied statistics, New York [u.a.]: Wiley, 1976.

[238] R. Cleveland and W. Cleveland, "Stl: A seasonal-trend decomposition procedure based on loess," *Journal of Official . . .*, vol. 6, 01 1990.

[239] F. Xiao, "Time series forecasting with stacked long short-term memory networks," *CoRR*, vol. abs/2011.00697, 2020.

[240] Z. Yu and G. Liu, "Sliced recurrent neural networks," 07 2018.

[241] L. Rokach and O. Maimon, *Data Mining with Decision Trees: Theory and Applications*. 12 2007.

[242] L. Cascone, S. Sadiq, D. S. Ullah, S. Mirjalili, H. Sidd, and M. Umer, "Predicting household electric power consumption using multi-step time series with convolutional lstm," *Big Data Research*, vol. 31, p. 100360, 11 2022.

[243] F. U. M. Ullah, A. Ullah, N. Khan, M. Lee, S. Rho, and S. Baik, "Deep learning-assisted short-term power load forecasting using deep convolutional lstm and stacked gru," *Complexity*, vol. 2022, pp. 1–15, 08 2022.

[244] philippe remy, "Stateful lstm in keras," 2016.

[245] O. Surakhi, M. A. Zaidan, P. L. Fung, N. Hossein Motlagh, S. Serhan, M. AlKhanafseh, R. M. Ghoniem, and T. Hussein, "Time-lag selection for time-series forecasting using neural network and heuristic algorithm," *Electronics*, vol. 10, p. 2518, Oct 2021.

[246] S. Glantz, "Primer of applied regression analysis of variance," 01 1990.

[247] T. Liu, Y. Guan, and Y. Lin, "Research on modulation recognition with ensemble learning," *EURASIP Journal on Wireless Communications and Networking*, vol. 2017, p. 179, 11 2017.

[248] A. Griewank, "Function space optimization," 2008.

[249] EEA, "Greenhouse gas emissions from transport in europe," 2023.

[250] R. Shankar, K. Chatterjee, and T. Chatterjee, "A very short-term load forecasting using kalman filter for load frequency control with economic load dispatch," *Journal of Engineering Science and Technology Review*, vol. 5, 03 2012.

[251] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into deep learning," 2021.

[252] R. E. Schapire and Y. Freund, *Boosting: Foundations and Algorithms*. The MIT Press, 2012.

[253] S. Elsworth and S. Güttel, "Time series forecasting using lstm networks: A symbolic approach," *ArXiv*, vol. abs/2003.05672, 2020.

[254] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer Science & Business Media, 1999.

[255] Z. Vujovic, "Classification model evaluation metrics," *International Journal of Advanced Computer Science and Applications*, vol. Volume 12, pp. 599–606, 07 2021.

[256] L. Rade and B. Westergren, *Mathematics Handbook for Science and Engineering*. Springer Berlin Heidelberg, 2004.

[257] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[258] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014.

Appendix A

LSTM - Gradients with Respect to Cell Parameters

The expressions $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_p}$ are identical to Equations 3.41 and 3.42, respectively. For the remaining cell parameters we have

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hc}} &= \sum_{t=1}^{T} \sum_{i=1}^{s} \frac{\partial \mathcal{L}}{\partial (\mathbf{S}_t)_i} \frac{\partial (\mathbf{S}_t)_i}{\partial \mathbf{W}_{hc}^{(t)}} \\
&= \sum_{t=1}^{T} \left((\mathbf{1} - \mathbf{S}_t^2) \odot \mathbf{i}_t \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{h}_{t-1}^{\top}, \\
\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xc}} &= \sum_{t=1}^{T} \sum_{i=1}^{s} \frac{\partial \mathcal{L}}{\partial (\mathbf{S}_t)_i} \frac{\partial (\mathbf{S}_t)_i}{\partial \mathbf{W}_{xc}^{(t)}} \\
&= \sum_{t=1}^{T} \left((\mathbf{1} - \mathbf{S}_t^2) \odot \mathbf{i}_t \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{x}_t^{\top}, \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_c} &= \sum_{t=1}^{T} \left(\frac{\partial \mathbf{S}_t}{\partial \mathbf{b}_c^{(t)}} \right)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{S}_t} \\
&= \sum_{t=1}^{T} (\mathbf{1} - \mathbf{S}_t^2) \odot \mathbf{i}_t \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t},
\end{aligned}$$

and analogously

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hf}} = \sum_{t=1}^{T} \left(\mathbf{f}_t \odot (\mathbf{1} - \mathbf{f_t}) \odot \mathbf{C}_{t-1} \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{h}_{t-1}^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xf}} = \sum_{t=1}^{T} \left(\mathbf{f}_t \odot (\mathbf{1} - \mathbf{f_t}) \odot \mathbf{C}_{t-1} \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{x}_t^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_f} = \sum_{t=1}^{T} \mathbf{f}_t \odot (\mathbf{1} - \mathbf{f_t}) \odot \mathbf{C}_{t-1} \odot \mathbf{o}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hi}} = \sum_{t=1}^{T} \left(\mathbf{i}_t \odot (\mathbf{1} - \mathbf{i_t}) \odot \mathbf{C}_{t-1} \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{h}_{t-1}^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xi}} = \sum_{t=1}^{T} \left(\mathbf{i}_t \odot (\mathbf{1} - \mathbf{i_t}) \odot \mathbf{C}_{t-1} \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{x}_t^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_i} = \sum_{t=1}^{T} \mathbf{i}_t \odot (\mathbf{1} - \mathbf{i_t}) \odot \mathbf{C}_{t-1} \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \left(\mathbf{o}_t \odot (\mathbf{1} - \mathbf{o_t}) \odot tanh(\mathbf{C}_t) \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{h}_{t-1}^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xo}} = \sum_{t=1}^{T} \left(\mathbf{o}_t \odot (\mathbf{1} - \mathbf{o_t}) \odot tanh(\mathbf{C}_t) \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t} \right) \mathbf{x}_t^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_x} = \sum_{t=1}^{T} \mathbf{o}_t \odot (\mathbf{1} - \mathbf{o_t}) \odot tanh(\mathbf{C}_t) \odot \mathbf{S}_t \odot (\mathbf{1} - tanh(\mathbf{C}_t)^2) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{C}_t}.$$

Appendix B

GRU - Gradients with Respect to Cell Parameters

The expressions $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_p}$ are identical to Equations 3.41 and 3.42, respectively. For the remaining cell parameters, we have

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hu}} &= \sum_{t=1}^{T} \sum_{i=1}^{s} \frac{\partial \mathcal{L}}{\partial (\mathbf{u}_t)_i} \frac{\partial (\mathbf{u}_t)_i}{\partial \mathbf{W}_{hu}^{(t)}} \\
&= -\sum_{t=1}^{T} \left(\mathbf{u}_t \odot (\mathbf{1} - \mathbf{u}_t) \odot \mathbf{h}_{t-1} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \right) \mathbf{h}_{t-1}^{\top}, \\
\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xu}} &= \sum_{t=1}^{T} \sum_{i=1}^{s} \frac{\partial \mathcal{L}}{\partial (\mathbf{u}_t)_i} \frac{\partial (\mathbf{u}_t)_i}{\partial \mathbf{W}_{xu}^{(t)}} \\
&= -\sum_{t=1}^{T} \left(\mathbf{u}_t \odot (\mathbf{1} - \mathbf{u}_t) \odot \mathbf{h}_{t-1} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \right) \mathbf{x}_t^{\top}, \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_u} &= \sum_{t=1}^{T} \left(\frac{\partial \mathbf{u}_t}{\partial \mathbf{b}_u^{(t)}} \right)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{u}_t} \\
&= -\sum_{t=1}^{T} \mathbf{u}_t \odot (\mathbf{1} - \mathbf{u}_t) \odot \mathbf{h}_{t-1} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t},
\end{aligned}$$

and analogously

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \left(\mathbf{r}_t \odot \mathbf{h}_{t-1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_t^2) \odot \mathbf{u}_t \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \right) \mathbf{h}_{t-1}^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \left(\mathbf{r}_t \odot \mathbf{h}_{t-1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_t^2) \odot \mathbf{u}_t \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \right) \mathbf{x}_t^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_h} = \sum_{t=1}^{T} \mathbf{r}_t \odot \mathbf{h}_{t-1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_t^2) \odot \mathbf{u}_t \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hr}} = \sum_{i=1}^{T} \left(\mathbf{W}_{hh}^\top \left(\mathbf{r}_t \odot (\mathbf{1} - \mathbf{r}_t) \odot \mathbf{h}_{t-1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_t^2) \odot \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_t} \right) \right) \mathbf{h}_{t-1}^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xr}} = \sum_{i=1}^{T} \left(\mathbf{W}_{hh}^\top \left(\mathbf{r}_t \odot (\mathbf{1} - \mathbf{r}_t) \odot \mathbf{h}_{t-1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_t^2) \odot \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_t} \right) \right) \mathbf{x}_t^\top,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_r} = \sum_{i=1}^{T} \mathbf{W}_{hh}^\top \left(\mathbf{r}_t \odot (\mathbf{1} - \mathbf{r}_t) \odot \mathbf{h}_{t-1} \odot (\mathbf{1} - \tilde{\mathbf{h}}_t^2) \odot \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{h}}_t} \right).$$

Appendix C

Layer Composition of Deep Learning Models

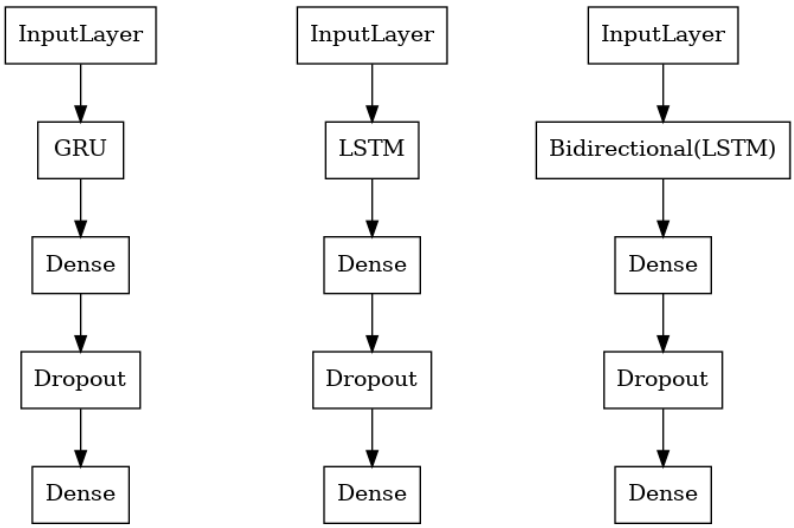


Figure C.1: Schematic overview of the GRU, LSTM and BiLSTM neural networks.

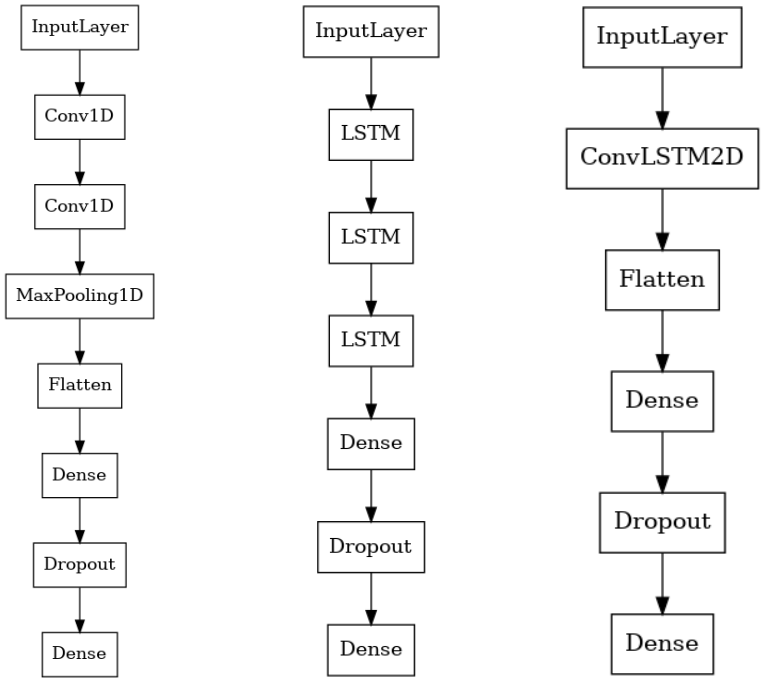


Figure C.2: Schematic overview of the Conv1D, Stacked LSTM and ConvLSTM neural networks.

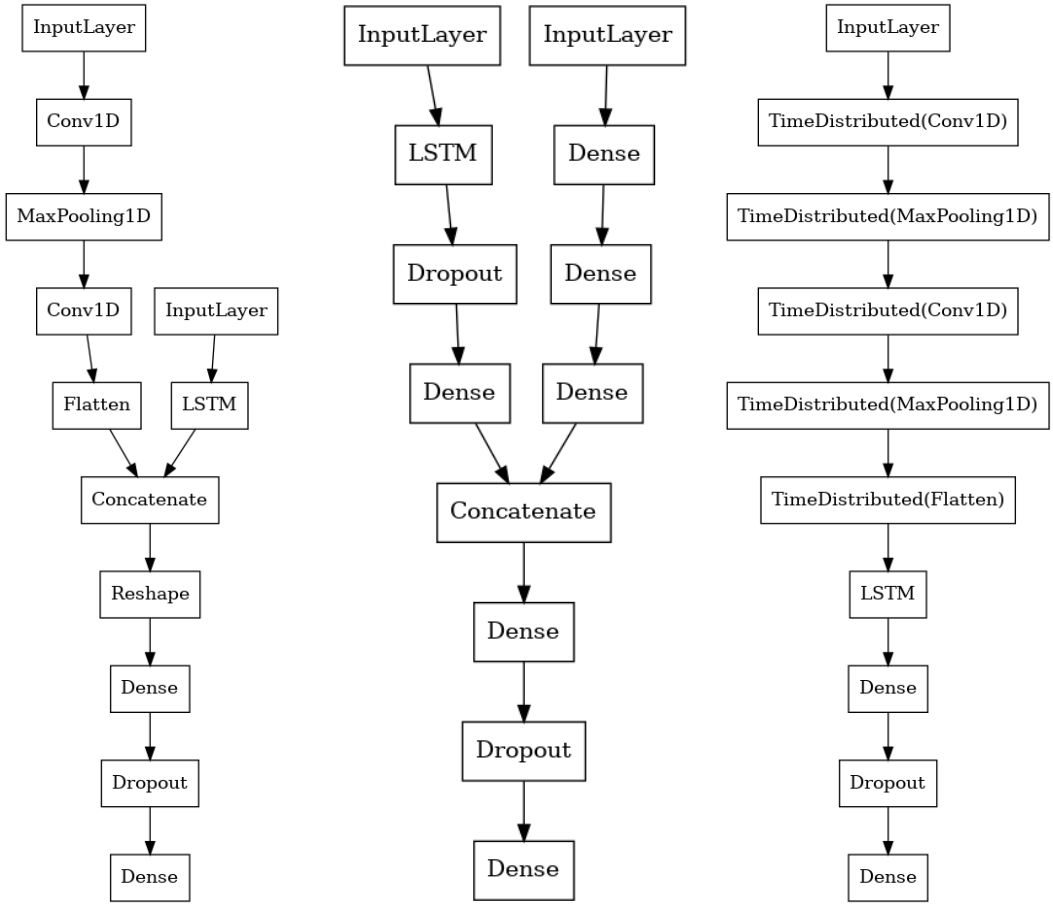
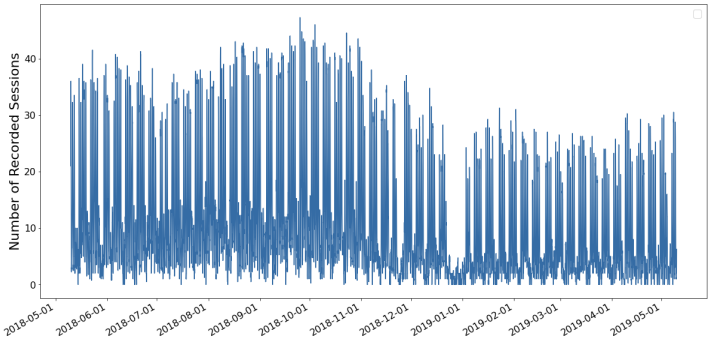


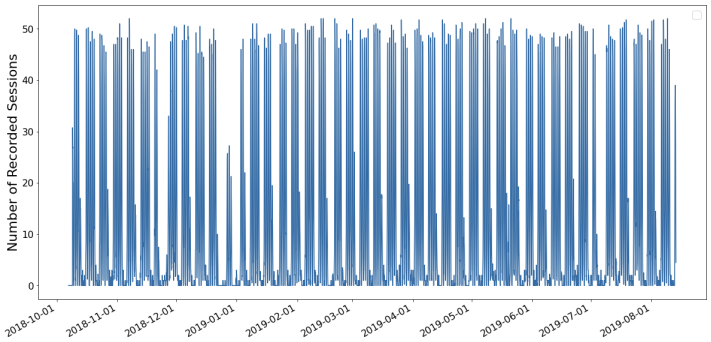
Figure C.3: Schematic overview of the PLCnet, Hybrid LSTM and CNN-LSTM neural networks.

Appendix D

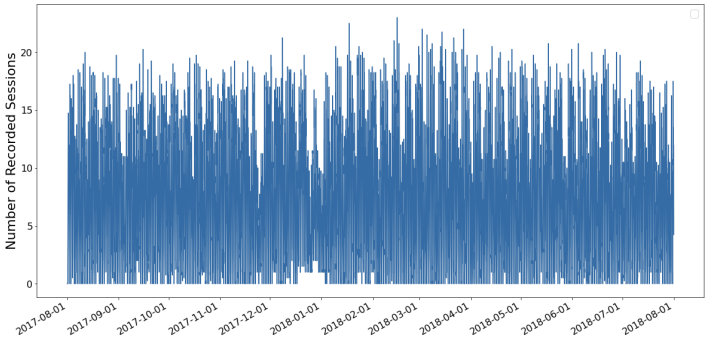
Aggregated Occupancy Time Series Plots



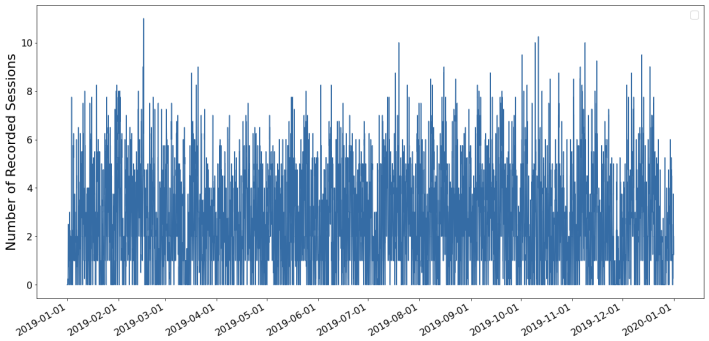
(a) ACN Caltech



(b) ACN JPL



(c) Palo Alto



(d) Boulder

Figure D.1: Aggregated occupancy state time series.