Bachelor's Thesis

A Case Study on LSTMs for Time Series Prediction

Tillmann Urs Brodbeck
Matriculation 964383
Bachelors's Program Cognitive Science
6th September 2019

First supervisor: Prof. Dr. Kai-Uwe Kühnberger
Institute of Cognitive Science
Osnabrück

Second supervisor: Mats L. Richter, M. Sc.
Agile IT Management
Osnabrück

Acknowledgements

I want to thank the people without whose help this work would hardly have been possible. Firstly, I want to thank Arvin Arora with whom this thesis was initiated, and who provided me with a lot of important technical feedback. Secondly, I want to thank my supervisors, Prof. Dr. Kai-Uwe Kühnberger and Mats Richter, for supporting me in writing this thesis and giving me helpful feedback. Thirdly, I want to thank the whole team of AIM for providing a welcoming atmosphere and for the support with technical issues. And last but not least, I want to thank my family and friends who supported me during this work intensive period: especially, I want to thank my parents, Maria and Walter, as well as Johannes, Lisa, Julia, Paul and Lukas.

Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

_____ _____

City, Date *Signature*

Abstract

This work analyses how well LSTM networks are able to make real world time series predictions. Currently, LSTMs are not the first choice when it comes to time series forecasting. However, an LSTM cells is an ANN, specialized in processing time sequences. The LSTM cell contains an internal memory state that is connected to its input and output through several gates. This enables sophisticated machine learning behaviour and allows the LSTM to understand the long term dependencies of the dataset. There are several possible ways of designing an LSTM architecture. This work explains the process of using multi-layered LSTMs,Sequence-to-Sequence architectures and LSTM-MLP hybrid models. These models are applied in several experiments and compared to an implemented MLP model that is currently used as an AIM prediction service. The results indicate that LSTM architectures can improve the prediction accuracy for a dataset and thus, LSTMs seem to be a valid choice when it comes to applied time series prediction.

Contents

List of Figures

List of Tables

Acronyms

ADAM	Adaptive Moment Estimation
AI	Artificial Intelligence
AIM	Agile IT Management
ANN	Artificial Neural Network
CMLP	Calendar Multilayer Processor Model
DL	Deep Learning
ELU	Exponential Linear Unit
LSTM	Long Short-Term Memory
LSTMD	Long Short-Term Memory Dense Model
MAAPE	Mean Absolute Arctangent Percentage Error
MAAPES	Mean Absolute Arctangent Percentage Error Shifted
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
ML	Machine Learning
MLSTM	Multi-layered Long Short-Term Memory Model
MLP	Multilayer Processor
MSE	Mean Squared Error
RNN	Recurrent Neural Network
SMLP	Simple Multilayer Processor Model
SS2S	Simple Sequence-to-Sequence Model
S2S	Sequence-to-Sequence Model
S2SD	Sequence-to-Sequence Dense Model

1 Introduction

1.1 Motivation and Objective

Due to the increased computational power that has become available over the last decade, particularly regarding high performance graphics cards, Artificial Neural Networks (ANNs) are being used more frequently in the area of Machine Learning (ML). Especially in the field of computer vision, ANNs, like the ResNet (He et al., 2016) and the Inception Networks (Szegedy et al., 2015), have shown the best performance in recent Kaggle competitions[1]. These competitions provide an recognised benchmark for ML algorithms. In the area of sequence learning, Recurrent Neural Networks (RNNs) are also becoming more popular. This is the case in the field of Artificial Intelligence (AI) game playing, such as competing against professional players in popular video games like Dota 2 (Summerville, Cook and Steenhuisen, 2016) and Starcraft II (Vinyals et al., 2017), and also in natural language processing, like the recent Contextual LSTM (Ghosh et al., 2016). There is an additional area, called time series prediction, that encompasses applications like stock market prediction, weather forecasting or, as in this case, predictive logistics. In this area of time series prediction, the classical ML algorithms, such as decision forests and ensemble methods, like the Extended Gradient Boosted Trees (Chen and Guestrin, 2016), are dominating in the field of sequence learning (Yi Jin, 2017).

However, what is currently used in economic time series forecasting applications is not publicly available in most cases. Therefore, this work explores the question of how ANNs perform in this applied field of sequence learning, to analyse, whether ANNs, including the Long Short-Term Memory (LSTM), are serious contenders when it comes to time series prediction. This work will focus on two special kinds of ANNs: the first model is the most common Deep Learning (DL) model, the Multilayer Processor (MLP). The second model class use LSTM cell architecture.

At Agile IT Management (AIM) in Osnabrück, Germany, an ANN is currently used for a time series forecasting service. It is being used in an application for a logistic load forecasting system that helps to optimize the resource usage of a production facility. The current prediction model is designed to only take meta information, i.e. calendar information, into account and to make a prediction for a single time point without considering the history of the time series. The model is an MLP and it will be explained in

[1] https://www.kaggle.com/competitions

Section 3.3.2.

Data scientists at AIM, were also able to achieve better predictions by adding information, such as a feature to represent the mean of the last n samples of the time series. It indicated that making the model consider the history of the time series might increase the prediction accuracy. The LSTM is a DL architecture that takes past information of the time series into account by design. This raised the scientific research question explored in this thesis: in comparison to an MLP model, could time series prediction accuracy be enhanced by using an LSTM based prediction model? It was hypothesized that this is true in the case of this real world dataset. The observations of the experiments used in this work provided the necessary information to answer this question.

1.2 Structure of this Work

This work will be structured as follows: Chapter 2 presents the underlying background knowledge and aids understanding of the algorithms used later. It covers the basic principles of the ANNs used, and the metrics for the model evaluation. Chapter 3 explains the methodology used. This section comprises the structure of the dataset and the implementation of the experiments, including data preprocessing, the predictor models and the evaluation methodology. In Chapter 4, the experiments will be listed and discussed. The various models will be evaluated and trained, and a test performance will be calculated. The final chapter is Chapter 5, which will form a conclusion of the results of this work and present future outlook.

2 Theoretical Foundations

This chapter will cover the foundational concepts upon which this work is based. The aim of this section is to provide any reader with enough background information about the models and learning algorithms used, to enable them to understand all the following chapters.

The first section covers the most basic ANN, the MLP, to provide the necessary foundation for the more complex recurrent models. The second section focuses on the capability of an ANN to learn based on its experience and Section 2.4 explains recurrent models, including the main focus of this work, the LSTM model. The final section comprises the metrics used for comparing the different models.

2.1 Multilayer Perceptron

IThis section begins with the quintessential ANN, which is the MLP. This approach was chosen because more advanced ANN such as the RNN and the LSTM are based on this model, and thus the MLP forms the basis for many commercial applications. Since they are currently used in services at AIM, MLPs will be used as benchmark models for the LSTMs in the context of this work.

The ability to solve tasks that are easy for humans to perform but difficult to describe in terms of a formal set of instructions Goodfellow, Bengio and Courville (2016) describe as the quintessential challenge to artificial intelligence. The MLP is a model that tries to solve this problem by adapting layers of internal parameters over the course of training using examples. This process is called supervised learning. The training examples are provided in the form of a dataset containing samples, which in turn consist of a feature vector, that is fed into the network. Then, the correct target vector is tried to be predicted by the network, when given the respective input feature vector.

The schematic view in Figure 1 shows the multiple layers that comprise the MLP; they comprise units of artificial neurons that resemble biological neurons. In an MLP every neuron of a layer is connected to every neuron of the previous layer, i.e a fully connected neural network. Each neuronal connection has a distinct weight that is held in a weight matrix w and it has an bias vector b. Thus, w and b represent the trainable parameters of a layer. From a vector x, which represents the output of the previous layer (or the input feature-vector of the MLP, if there is no previous layer), the activation $wx + b$ can be calculated. Finally the layers have a certain nonlin-

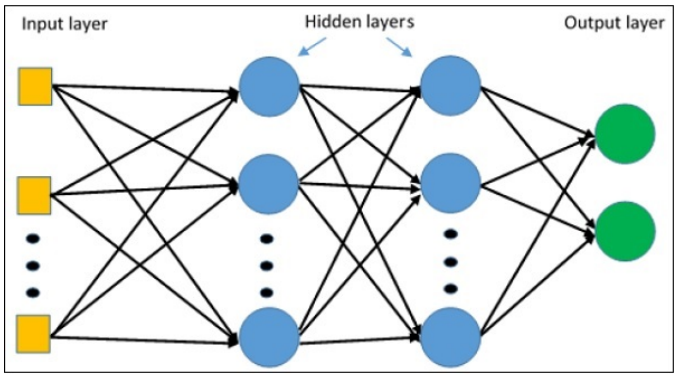

Figure 1: Exemplified schematic view of a MLP; retrieved from `https://www.tutorialspoint.com/tensorflow/tensorflow_multi_layer_perceptron_learning` (visited on 08/06/2019).

ear activation function a that is applied to the activation on every neuron it contains. A resulting vector $y = a(wx + b)$ can be calculated with a vector size according to the amount of artificial neurons of that layer. Generally speaking, each layer can be understood as a representation that impacts the representation of the subsequent layers, together with the parameters between these layers.

Hence, this forms an arbitrary complex computational graph or a pipeline that is used to transport information from the input to the output, for making predictions, and from the output to the input, for the parameter training (Section 2.3). For example, a prediction is made by calculating the activations of the subsequent neurons layer by layer, starting with the input layer and finishing with the output layer of the MLP. The input vector of the MLP contains the values of the features the network perceives and the output vector contains the values of features it predicts.

The training of such an algorithm is termed deep learning because of the deeply nested sequential instructions that have to be executed to evaluate the architecture (Goodfellow, Bengio and Courville, 2016). In addition to many possible parameters, a seemingly simple MLP comes with a plethora of possibly changeable hyperparameters. There are different activation functions to choose from and the whole architecture can be modified, including the amount of layers, the form of the input and output vectors and the size of the layers. Furthermore, this list can be extended to include the preprocessing and training hyperparameters.

To be able to develop models and adapt these parameters in a reasonable

manner, an algorithmic approach is necessary to evaluate the performance of a model. This topic is covered in the following section.

2.2 Model Evaluation

Many parameters can be modified in the models and therefore, there are many different variants of models to compare. Because there is this large search space of model variations, a sophisticated approach to model comparison is needed. For this model comparison a metric must first be chosen. A metric provides a measure of how well the algorithm performs according to its specific criteria. Due to the fact that the task in this case is forecasting, the metric should describe how similar a prediction is to an actual observation. Different metrics are applied in the context of this work. The Mean Absolute Error (MAE), Mean Squared Error (MSE), Mean Absolute Percentage Error (MAPE) and Mean Absolute Arctangent Percentage Error (MAAPE) will be listed in the following sections.

2.2.1 Mean Absolute Error

The MAE is a metric that calculates the absolute differences at every timestep t between the predicted data point y_t and its actual value, or target value, a_t and takes the mean value of these differences. (Sammut and Webb, 2010a) For a time series of length n we get the following equation:

$$MAE = \frac{\sum_{t=1}^{n} |a_t - y_t|}{n} \tag{1}$$

2.2.2 Mean Squared Error

Instead of the absolute values in the MAE, the MSE calculates the squares of every pair of x_t and y_t and then similarly, the mean of these pairs is then calculated (Sammut and Webb, 2010b):

$$MSE = \frac{\sum_{t=1}^{n} (a_t - y_t)^2}{n} \tag{2}$$

In particular, a comparison of the MAE and MSE give interesting insights, e.g. outliers are punished more strongly for the MSE. Comparing both can be used to track how stable the training is, i.e. how similar the two metrics stay during the training of a neural network.

2.2.3 Mean Absolute Percentage Error

In contrast to the MAE and MSE, the mean absolute percentage error MAPE gives a relative estimation of the error. To calculate the MAPE, the mean of the division of the differences between a_t and y_t divided by a_t is taken. It returns a value between 0 and 1 for every difference as a percentage measure of how well a_t is approximated compared to its actual value. This can be multiplied by 100% and thus as a percentage measure it enables scale-independence and a more intuitive interpretability than the MAE and the MSE. Additionally, the values of the $MAAPE$ can be compared with each other when evaluating the performance of predictions regarding different datasets.

$$MAPE = \frac{\sum_{t=1}^{n} |\frac{a_t - y_t}{a_t}|}{n} \tag{3}$$

According to Myttenaere et al. (2015) the conventions for the case $a_t = 0$ are such that $\frac{0}{0} = 1$ and if $y_t \neq 0$, $\frac{y_t}{0} = \infty$. In general, the MAPE has the problem of calculating infinites for values close to zero.

2.2.4 Mean Arctangent Absolute Percentage Error

To solve the problem of calculating infinites with the MAPE there is an adapted version of the MAPE: S. Kim and H. Kim (2016) have introduced the MAAPE. This metric simply applies the arc tangent tan^{-1} function to each of the relative differences of the MAPE:

$$MAAPE = \frac{\sum_{t=1}^{n} tan^{-1}(|\frac{a_t - y_t}{a_t}|)}{n} \tag{4}$$

This yields a metric that more robustly resists large errors and thus it will be the main metric used for the model comparison of this work.

Currently at AIM a small variation of the MAAPE is used. Instead of having a_t in the denominator of the arc tangent the value is shifted up by 1. This avoids divisions by 0 because the dataset often has zero values as targets.

$$MAAPE_s = \frac{\sum_{t=1}^{n} tan^{-1}(|\frac{a_t - y_t}{a_t + 1}|)}{n} \tag{5}$$

For the experiments in Chapter 4 the decision was made to use the Mean Absolute Arctangent Percentage Error Shifted (MAAPES) as the main metric. Still, both variations of the MAAPE were observed while experimenting to enable more refined model comparisons.

The neural network can be trained after an error metric for its prediction has been defined. This will be covered in the following section.

2.3 Training Neural Networks

In the case of the forecasting application developed in this work the objective is to create a prediction for a future time point with a minimal error according to a specific metric. This specific metric in the context of training is then called a loss function L. Since there are different ways of measuring the deviation of a predicted and observed time series, different metrics can be used for L, as described in the previous section. Because we have many trainable parameters for the neurons, as mentioned in Section 2.1, every single parameter can be adapted by calculating the derivative of that parameter in respect to L and then adapting the parameter in the direction of the negated derivative. This procedure is called stochastic gradient descent. Because after the parameter adaptation L is likely to return a smaller value for a future evaluation of the network.

There are complex algorithms to calculate the exact parameter update - these are called optimizers. An overview of optimizers is presented by Ruder (2016). In the experiments included in this work the Adaptive Moment Estimation (ADAM) optimizer (Kingma and Ba, 2014) has been chosen for several reasons. Firstly, it is the most prominent one in the state-of-the-art. DL (Bianchi et al., 2017);and secondly, it typically requires relatively little tuning according to Kingma and Ba (2014). Having reliable hyperparameter choices helps to reduce the complexity of the experiments in Section 4. Thirdly, it has already been used for the MLP at AIM and thus, it helped making the experiments more comparable.

Due to the fact that the main goal of Machine Learning is to calculate accurate predictions for unseen data, it is not enough to simply optimize a network, to calculate the right outputs to the respective features in the data that the model is trained on. The main goal is to make the network learn the underlying patterns in the data, i.e. to be able to generalize from it and eventually predict unseen data correctly. There are two main methods to support the learning capability of neural networks. The first method is to split the data into distinct sets, so that the model can be evaluated on a subset of the data it was not trained on. This is more broadly discussed in Section 3.4. The second method to improve learning is the use of so-called regularisation techniques. Regularisation generally reduces the vulnerability of the model to overfit, i.e. the overly strong adaptation to the training data.

For regularisation dropout will be used, which is a computationally inexpensive, yet effective, method for regularisation. Furthermore, dropout is controlled by a dropout rate parameter that steers how many units of the preceding layer to the dropout layer are multiplied with zero, to temporarily

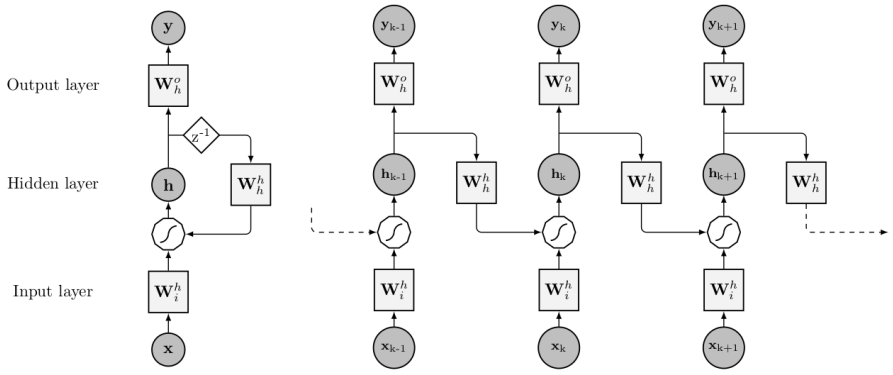

Figure 2: This is an is a schematic view of an RNN (Bianchi et al., 2017). On the left side in the folded form (showing the recurring modules for all time steps) and on the right side in the unfolded form (showing every time step individually). k represents one time step, the squares W_i^h, W_h^h and W_h^o represent the input, hidden and output weights, respectively. The white circle signifies the activation function and z^{-1} is a time delay operator.

deactivate their effect during training time. This leads to a different set of active neurons in the network for every training run and reduces the risk of overfitting the data.

Now that a basic understanding about ANNs has been provided, more complex types of DL models will be introduced in the next section.

2.4 Recurrent Neural Networks

This section will cover the foundations of a special type of ANN, the RNN. The first subsection will cover the theory of a simple RNN. Following that, Section 2.4.2, will focus on the explanation of LSTMs and detail its advantages over the simple RNN. The goal of the final subsection is to explain a special type of RNN model with an encoder-decoder architecture that is used in this work.

2.4.1 Simple Recurrent Neural Network

In a classical MLP there are no feedback connections, which is why it is also considered a feedforward neural network. If we extend the MLP to also allow for feedback connections, these networks are then called RNNs (Rumelhart, Hinton, Williams et al., 1988). Specifically, an RNN is a network that spe-

cializes in processing sequences; the time aspect of data is modeled by cycles in created inside the network (Patterson and Gibson, 2017). The cyclical structure can be seen in Figure 2 on the left. In general, these are extended ANNs, mostly used for the use cases of time series forecasting or natural language prediction.

The simple RNN can be compared to an MLP with three layers, an input, a hidden and an output layer. As shown in Figure 2, the weight matrix W_i^h connects the input to the hidden layer, while the weight matrix W_h^o connects the hidden to the output layer, like a normal feedforward neural network also would. Additionally, there is the recurrent connection from the hidden layer to itself in a subsequent time step. This connection is parameterized with the weight matrix W_h^h.

When applying the gradient descent algorithm to an RNN model, the parameter updates are computed analogous to the MLP. This can be done, when taking the unfolded form of the RNN, which is exemplified by Figure 2 to the right. For all time points t in the series the loss L_t is computed for the predictions y_t. The Jacobian, i.e. the partial derivative over all values of a weight matrix W in respect to every L_t of y_t, is calculated, in order to compute the gradient update of W. These Jacobians are then added up to produce update values over the whole span of y_t and the gradient descent is then applied in the same way as for the MLP. This procedure is termed Backpropagation Through Time (Bianchi et al., 2017).

However, this calculation has one quirk; because the calculation of the Jacobians, or gradients, is done through the repeated application of the chain rule. Depending on the activation function, this leads to the issue of vanishing or exploding Jacobians according to (Patterson and Gibson, 2017). For the usual nonlinear activation function, the Jacobian calculated in respect to L_t becomes smaller for every layer of the unfolded RNN it passes in this backpropagation procedure. It becomes smaller due to the fact that the slope of most activation functions is usually close to zero. Because of the small gradients and the repeated application of the chain rule, the parameter updates in time step t_0 are mainly determined by the losses L_{t_0+1}, L_{t_0+2}, L_{t_0+3}... of the next few outputs, but the further in the future the y_t is, the smaller the gradient and thus the update will be. Generally speaking, long term dependencies are rather difficult to learn for the RNN. This is commonly known as the vanishing gradient problem and it is a very common problem for RNNs. It is also one of the main motivations why the LSTM cell was developed, which will be covered in the next section.

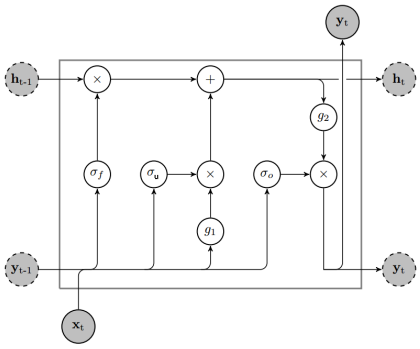

Figure 3: Illustration of an LSTM cell (Bianchi et al., 2017): x represents input and y output. Dark grey circles with a dashed border represent the internal state variables, the content of which is passed along the cells. g_1 and g_2 are non-linear transformation. $+$ and $\times$ represent linear operations. While σ_f, σ_u and σ_o are the sigmoid functions used for the forget, update and output gates, respectively.

2.4.2 Long Short-Term Memory Network

The LSTM, as introduced by Hochreiter and Schmidhuber (1997), falls into a subcategory of RNNs, called gated RNNs. In contrast to the simple RNN cell, a LSTM cell propagates two states to a subsequent cell: its own output y_t and the hidden state h_t of the LSTM, which is also called cell state. (Bianchi et al., 2017) The hidden state h_t is controlled by three gates: The forget gate, the update gate and the output gate. All of those gates contain trainable sigmoid activation functions $\sigma(x)$, that are defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{6}$$

These $\sigma(z)$ inside the gates serve as a mask with values in the range of 0 and 1 that are elementwise multiplied onto another matrix and thus serve as an information filter. This results in three filters, one for each gate. In detail, the calculation of every filter is similar: the current input x_t is connected through the weight matrix W, while the previous cells output y_{t-1} is connected through the weight matrix R. The two inputs are added up with a bias b resulting in the following equations, where f, u and o denote the components of the respective filters:

$$\text{forget filter: } \sigma_{f,t} = \sigma(W_f x_t + R_f y_{t-1} + b_f) \tag{7}$$

$$\text{update filter: } \sigma_{u,t} = \sigma(W_u x_t + R_u y_{t-1} + b_u) \tag{8}$$

$$\text{output filter: } \sigma_{o,t} = \sigma(W_o x_t + R_o y_{t-1} + b_o) \tag{9}$$

In the following will be explained how these filters are applied inside the LSTM cell: firstly, the hidden state is adapted by the forget and the update gates. The application of the forget filter is a simple elementwise

multiplication of $\sigma_{f,t}$ to h_{t-1}. Before the update can be applied, the LSTM first calculates the update candidates c_t. The values for c_t are provided by the nonlinear activation function g_1. Analogous to the filters, g_1 calculates the sum of the input x_t weighted with a matrix W_c, y_{t-1} with a weight matrix R_c and an additional bias b_c. This results in the following equation for the update candidates:

$$c_t = g_1(W_c x_t + R_c y_{t-1} + b_c) \tag{10}$$

c_t is then filtered by elementwise multiplication with $\sigma_{u,t}$ and the output is then added to the output of the forget gate resulting in the new hidden state:

$$h_t = \sigma_{f,t} \odot h_{t-1} + \sigma_{u,t} \odot c_t \tag{11}$$

Lastly, a second nonlinear activation function g_2 is applied to h_t. For g_1 and g_2 the hyperbolic tangent activation functions are usually used. The resulting vector is then elementwise multiplied by $\sigma_{o,t}$ resulting in the new output:

$$y_t = \sigma_{o,t} \odot g_2(h_t) \tag{12}$$

Therefore, when training an LSTM, only the parameters of the gate functions are adapted. Due to this improved RNN architecture, the issue of vanishing gradients does not occur as it does with the simple RNN. The main reason for the resolution to this issue, according to Bianchi et al. (2017), is that there are no nonlinear transfer functions applied to the transfer of h_{t-1} to h_t. Instead, the transfer function, is the identity function and, therefore, the contribution of past states to a gradient calculation remains unchanged over time.

All in all, the capacity to memorize long term dependencies of an LSTM is greater in comparison to other RNN architectures. This is an important property for time series forecasting, because the LSTM is capable of learning large scale patterns in the data. Nevertheless, the content of a cell cannot remain completely unchanged over time, therefore it has to be accepted that some information will be forgotten over time. The experiments for Chapter 4 will try to find the right balance between having a sufficient number of parameters in W and R, while simultaneously avoiding overfitting. The next section presents another variation for connecting multiple RNNs in a Sequence-to-Sequence Model (S2S) architecture.

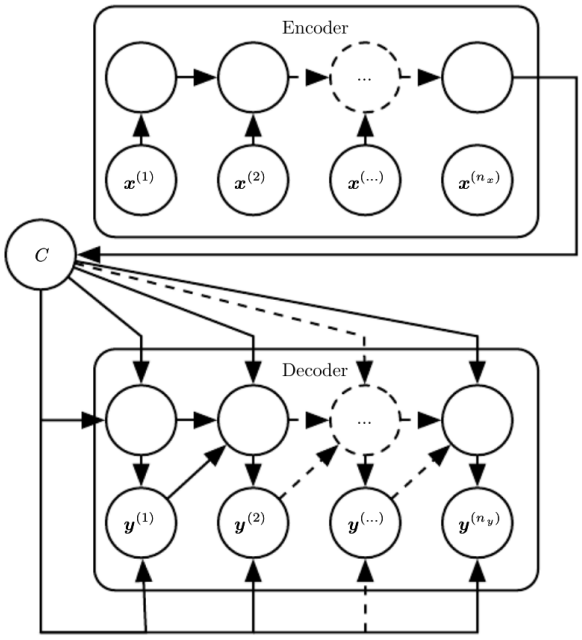

Figure 4: Schematic example of an S2S architecture (Goodfellow, Bengio and Courville, 2016). An unfolded view of the RNN cells is shown. The input is a sequence of length n_x creating the context C, from which the output sequence y of length n_y is created. C can, as the bottom arrows indicate, also directly influence y.

2.4.3 Encoder-Decoder Sequence-to-Sequence Model

The encoder-decoder architecture was independently introduced by Cho et al. (2014) and Sutskever, Vinyals and Le (2014) for natural language translation. It consists of an encoder, i.e. an input RNN, and a decoder, i.e. an output RNN. The main motivation for the development of encoder-decoder architectures was that one can map an input sequence to an output sequence of different lengths. This is also an important property needed for the application of AIM, as well as being required by many other applications in the area of machine translation, speech recognition or question answering. (Goodfellow, Bengio and Courville, 2016)

The encoder RNN processes the input sequence and emits a so-called context vector C instead of an output sequence. C is usually a simple function of the final hidden state h_{t_x} of the encoder RNN. The decoder is a type of RNN that does not receive an actual input sequence. Instead, it either receives the vector C, as an input for every time step or its hidden state is initialized using C, or both. For reference see Figure 4. In principle, there are many more possibilities for defining the encoder or the decoder and how to connect them, some of these variants will be further explained in Section 3.3. It is included in the next chapter, that will cover the methodology of

this work.

3 Methods

After covering the theoretical foundations in the previous chapter, this chapter will focus on the actual methodologies and implementations used in this work.

3.1 The Dataset

To specify exactly what this work tries to accomplish, it is important to first take a deeper look at the dataset: the dataset itself comprises observations of a real-world production facility and was retrieved using the Enterprise-Resource-Planning software SAP[2]. The dataset is provided by the customer, i.e. the enterprise behind the production facility itself, and thus it is difficult to analyse how the data exactly was retrieved. Therefore, there could be several impairments in quality, affecting its representability of the actual load. Furthermore, for this dataset it was not declared how homogenous measurements where taken or whether the measurements were biased. The

[2]https://www.sap.com

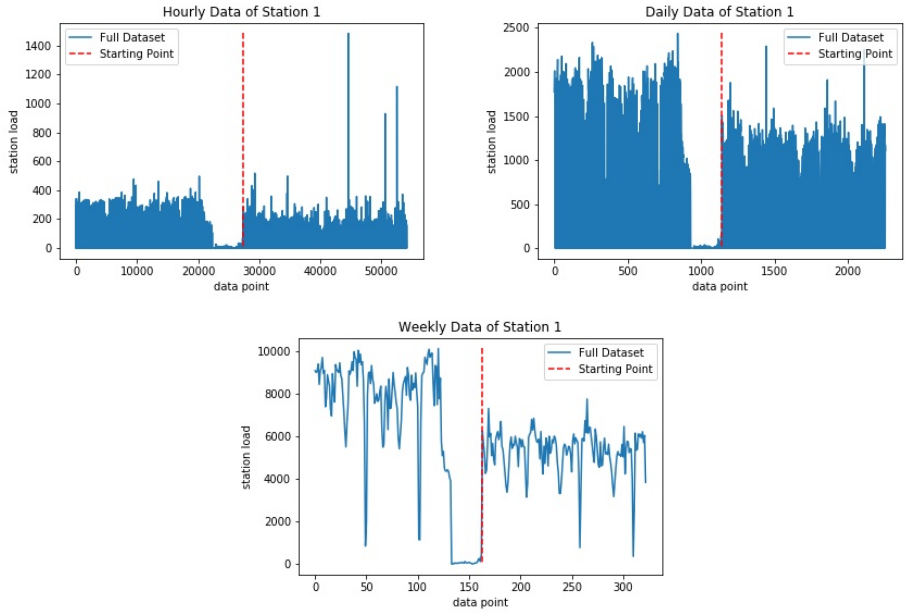

Figure 5: Example of the dataset of Station 1: The measurements get more stable after February of 2016, where the red cutting line is located.

data format is provided in integer values representing the number of incoming production tasks in a time period. This was measured for several stations inside the facility, so this is termed the station load. Furthermore, it is measured in three granularities of time at weekly, daily and hourly intervals.

The dataset contains n integer features, that represent the load of the station s_n of the facility. In total we have $n = 21$ stations provided in the dataset. These can be divided into three subsets. The first ten stations s_{1-10} are the stations relevant for the prediction. Their past values are used as inputs for the models of this work. The second five stations s_{11-15}, would have been more difficult to predict, since their initial start-up was later than the initial start-up of the first ten stations. The last six stations, s_{16-21}, provided additional information, but were not required to be predicted, therefore, they were included as input features for the models only.

The total measurements began in January of 2013 and ended in March of 2019. But since the dataset is not coherent before February 2016, earlier data was dismissed for this project. The point where the data was cut is illustrated in Figure 5.

In addition to the time series, the dataset was enriched with calendar information. In total, there are eleven integer features shown in Table 1, while CW is the calendar week and H signify holidays. *Number of H* counts the holidays in the current time period, *Since last H* and *To next H* measure the distance in days to the respective H in the future or past. Then, there are two rolling windows, one forward and one backward. In these windows, the number of holidays are summed up and represented as an integer value. This calendar dataset is designed is such a manner, because on holidays or on weekends the station load regularly is low.

Furthermore, the requirement for the service of AIM was to predict a certain range of data points for every time point put in. This output window $s_{outwindow}$ was different for each one of the data categories. For the hourly prediction, the models outputted 24 hours into the future. For the daily prediction, $s_{outwindow}$ was 5, i.e. one working week. For the weekly prediction, the models predicted the next 4 weeks for every time point.

	Year	CW	Month	Day	Weekday	Hour	No. Hs	Since last H	To next H	Roll. H fw	Roll. H bw
Hourly	+	+	+	+	+	+	+	+	+	+	+
Daily	+	+	+	+	+	–	+	+	+	+	+
Weekly	+	+	+	–	–	–	+	+	+	+	+

Table 1: Categories of the calendar dataset. + signifies that the feature is used for the time category and – signifies that the feature was not used for the respective category. *Roll.* is rolling, fw is forward and bw is backward.

Having now explained the basic structure of the dataset and the prediction requirements, in the following section the preprocessing of this work will be described in detail.

3.2 Preprocessing

Before applying the models, a certain amount of preprocessing was done. Firstly, it was tried to detect outliers, visible in Figure 5 and replaced their values with the mean value $mean(s_n)$ of the corresponding s_n. Due to the fact that the data is bound from below by zero, only outliers in the upward direction could be removed. Every value with a value higher than $mean(s_n)+(std(s_n)*tol)$ was categorised as outlier, where tol denotes a scalar value for the tolerance of the outlier removal. Analysing the curve for several the outlier removals, it was possible to establish that $tol = 4$ for the weekly and daily measured curves and $tol = 15$ for the hourly measured curve where sufficient.

Then, data points were removed that were outside the working hours of the facility. Therefore, any weekends were removed and hours earlier than 6am and after 21pm were removed.

To improve the learning capability of the models, the data points were scaled into the interval $[0, 1]$. This was done for the three granularities g of the data, before they were applied the a models. Furthermore, first the minimum $min(g)$ and the maximum $max(g)$ of the category were calculated and then

$$v_{scaled} = \frac{v - min(g)}{max(g) - min(g)} \tag{13}$$

was applied to every value v of the dataset g. Through applying this scaling, the minimum value of the dataset was set to 0 and the maximum value of the dataset was set to 1.

Finally, the dataset needed to have the right structure to be compatible with the time-dependent LSTMs models and the benchmark models. Additionally, the future calendar information should be available for the predicted time point for the models, while the future time series itself should obviously be not visible. Therefore, three individual matrices were created, that are used for the training of the models. These three matrices have one individual axis, that has the size of the amount of features represented. While the second axis, containing the different time points has to be equal: the first matrix x is the input matrix. It comprises the number of samples in the dataset s_{data} minus the size of the output window: $s_{data} - s_{outwindow}$. Additionally, a target matrix a and the calendar matrix c were created with

the size $s_{data} - s_{outwindow}$. The amount of features in c were the calendar features for every day in the prediction window appended.

3.3 The Models

This section will explain what models have been used for the experiments. First, the benchmark models will be explained, which include two naive predictors and the two versions of MLPs. And secondly, different configurations of LSTM models have been used.

3.3.1 Naive Predictors

First of all, naive predictors were used to estimate the general performance of the algorithms. Therefore, two kinds of naive predictors that are relatively easy to interpret were chosen, so they give a useful baseline of how good a model should perform.

The first naive predictor simply takes the previous day's value and uses this exact value as a prediction for the current day.

The second naive predictor remembers all the previously observed values and calculates their mean value. This mean is then used to predict the current day.

In addition to the naive predictors, the models currently used at AIM were used as benchmarks. These are described in the next section.

3.3.2 Current Production Model

The current production model is an adaptation of the model, that is already used at AIM for the time series prediction task. Therefore, this is the main benchmark model of this work. It consists of six layers: In the input layer only the calendar information about the days to predict were fed in. This input layer is fed into four consecutive layers of respectively 435, 517, 132 and 126 neurons with an Exponential Linear Unit (ELU) activation function (Clevert, Unterthiner and Hochreiter, 2015),

$$ELU(x) = \begin{cases} x & \text{for} \quad x > 0 \\ \alpha(e^x - 1) & \text{for} \quad x \leq 0 \end{cases}, \tag{14}$$

where α is a scale factor for negative inputs, in this case simply $\alpha = 1$ is used. The parameters of this layer are initialised according to a Glorot uniform initializer (L. S. Kim, 1993). Which comprises the drawing of samples from

a uniform distribution within a range $[-l, l]$, with the limit l calculated as

$$l = \sqrt{\frac{6}{n_i + n_o}}, \tag{15}$$

and with n_i being the number of input units of the weight matrix and n_o being the number of output units of this matrix. After that, a consecutive fully connected neural layer, also termed densely connected layer, was appended. Then, there is an optional dropout layer. This is done because Keras Library (Chollet et al., 2015) dropout is implemented as its own layer that can be put right after a normal densely connected neural layer. Finally, the output of this layer is fed into a final dense layer, generating the predictions y without another nonlinear activation function.

An interesting fact about the MLP of AIM is that it does not receive the time series as an input at all and only uses the calendar dataset as input. Advantages of this architecture are that it is faster, requires less resources and can be used to predict any time point in the past or future, without requiring any information about the timeseries for the prediction. To make it more comparable to the LSTM, an extended variant of the MLP was implemented that also receives the load information of the stations of the n_x previous time point concatenated to the calendar data of the current time point, thus, n_x being the size of the input window. Thus the structure of the input features for the MLP and the LSTM models are equal. For every parameter not worth mentioning here explicitly, the default parameters are used of the Keras library and thus can be found in the Keras documentation (ibid.). These models are then used as a benchmark for the research models discussed in the following section.

3.3.3 Multi-layered LSTM Model

This work focuses on analysing LSTMs, so several models using the LSTM cell architecture with an LSTM cell configured as presented in Section 2.4.2 were implemented. For the activation functions g_1 and g_2 the hyperbolic tangent is used as it was also described in Section 2.4.2. As the initializer for input weights W the Glorot uniform initializer is used in the same way as for the MLP, and an orthogonal initialization is used for the recurrent weights R. To provide more detail, this is the generation of a random orthogonal matrix. An orthogonal matrix is defined by all column vectors of that matrix being orthonormal, i.e. having the length of 1 and having a scalar product of 0 with every other column vector (Saxe, McClelland and Ganguli, 2013). In

addition to the cell a dropout parameter is applied that steers the fraction of temporary deactivated units for the linear transformation of the inputs.

Then, four distinct variations of LSTM models were implemented. The first variation is a simple multi-layered LSTM model. It receives the concatenated input of x and c, i.e. $inp = x \oplus c$ and inp gets fed into an LSTM cell with the hidden state size $size_h = size_a$ according to the amount of features of a. This LSTM cell returns an output sequence seq_n and hidden state h_n, while t is the amount of time points of inp. seq_n is then fed back into the equivalent model initiated with h_n. This is repeated for $n-1$ amount of times, thus $n-1$ equals the number of layers. Lastly, seq_{n-1} and h_{n-1} are fed into the last LSTM cell that returns a final sequence seq_n, that also is the output of the model. The idea is that this model has different layers of LSTMs, just like the different layers of an MLP. The first LSTM might extract simple features, whereas the subsequent LSTMs creates more complex features from the encodings of the previous versions. This model architecture will be termed as Multi-layered Long Short-Term Memory Model (MLSTM) for the further context of this work.

3.3.4 Encoder-Decoder Model

A second model, that was implemented is an encoder-to-decoder architecture. The encoder receives the x matrix and encodes it into an LSTM with $size_h = size_a$. This cell then returns its final hidden state h_{enc} and output state y_{enc}. These states are forwarded into a decoder LSTM with its states initialised with h_{enc} and its output state initialised with y_{enc}. Therefore, the decoder has the same $size_h$ as the encoder. In contrast to the multi-layered LSTM model of the previous section, this subsequent cell receives c as a distinct input in the decoder. The output of the decoder is thus the output of the model.

3.3.5 LSTM Dense Models

Additionally, the two previous models were implemented in a variation that uses a dense layer as the final layer. Therefore, output of the LSTM is not used directly as the output of the model. Instead it can be transformed into the right size due to the final dense layer. The advantage of such an architecture is that it allows LSTMs to have greater values for $size_h$. This is particularly useful when a basic LSTM model is not able to recognize enough patterns in complex data. Because the Long Short-Term Memory Dense Model (LSTMD) provides more neural units per layer, this model

might be a better choice in that scenario.

In this work, this architecture is applied to the MLSTM model, forming an LSTMD model where the amount of layers and the size of the hidden state can be adapted. Also this model was applied to the S2S, forming an Sequence-to-Sequence Dense Model (S2SD) model. To distinguish the archtectures, the model of Section 3.3.4 will be termed Simple Sequence-to-Sequence Model (SS2S).

3.3.6 The Training Configuration

The MAE is used as the metric to calculate the loss of the backpropagation of the ANNs in this work. This metric was chosen over the MSE because it is less sensitive to remaining outliers and because ANNs do not gain an additional advantage by using percentage errors as the MAAPE. Still, the all metrics will be used, when analysing the validation curves.

Furthermore, the ADAM optimizer has been used with the default hyperparameter setup of Keras[3].

3.4 Evaluation Methodology

Now that the implemented models have been covered, this chapter will focus on the methodology used to evaluate and compare the models. In the following two sections, the creation of the training set, the validation set and the test set will be described.

3.4.1 Creating the Test Set

Before starting to apply the models, the dataset was split into two parts: the first part comprising the first 85% of the data and the second part the most recent 15% of the data.

The first part is used for training and validation. Training is the process of fitting the parameters according to gradient descent explained in Section 2.3, and validation is an evaluation phase after a certain number of training steps, to track the training performance during training time. The evaluation frequency in Keras is set to one epoch, i.e. the process of training once on every sample of the training subset.

And the second part is taken aside and will be used as a test set after selecting the most promising models and parameter configurations. This test set will only be used once in the final evaluation of this work. Thus, it is

[3] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

ensured that the final performance evaluation is as close as possible to the performance when using the model as a real-world prediction service, predicting data points it has never seen before. There are multiple possibilities of splitting of the data into the training and validation set. These are covered in the next section.

3.4.2 Creating the Training and Validation Sets

In general, there are different ways of splitting the larger part of the dataset into training and validation set. A common way is to give the validation set the same size as the test set. In this case, one would reserve another 15% for the validation set resulting in a 70-15-15 split. Then we would train every model for 70% of the total timesteps and validate it on the following 15%. This is the way the training and validation sets for the experiments were split.

4 Results and Discussion

In this chapter will describe the entire methodology that was applied, whilst experimenting with the LSTM models. Goodfellow, Bengio and Courville (2016) delivered a theoretical framework on how to successfully apply DL techniques. It is described as the four central steps of deploying a model:

1. An error metric and a goal value you want to achieve with your models have to be determined.

2. A working model has to be established as soon as possible.

3. The properties and the performance of the model should be analysed with according to the metrics. The property of the model to generalize for new data, as it should, have to be modeled, and defects in your data or algorithms have to be detected.

4. The last step, according to Goodfellow et. al., is the parameter search. Repeatedly incremental changes should be made, such as gathering new data, adjusting hyperparameter, or changing algorithms, based on specific findings from the instrumentation used. (ibid.)

Because for the main metric is was decided to use the MAAPES, the next step would be the analysis of the performance of the benchmark models. Thus, a target value for the research models can be estimated. This will be done in the first subsection. The working models were established in the course of this work, already. The following sections will analyse how these models behave and the best hyper parameters will be searched. First the experiment methodology and the first parameter search are explained in Section 4.2. Then the analysis will be discussed in Section 4.3. In the sections after that, hyper parameters options will be analysed. Lastly, in the two final sections, the test performance is derived and the final results are presented.

4.1 Analysing the Naive Predictors

To determine the minimum target values that should be reached, the naive predictors, introduced in Section 3.3.1, were analysed. Because there are three versions, hourly, daily and weekly, of the dataset, each of the mentioned benchmark models for each version of the dataset was evaluated. As it is presented in Table 2, the error metrics of the previous predictor yields a value that is close to the performance of the models for the smaller weekly

datasets. On the other hand, the mean predictor does not show comparable results. In the table, the mean predictor always shows large error values in comparison to other models. Therefore, in the later analyses only the naive mean predictor will be included as reference.

4.2 The Design of the Comparison

Knowing the metrics for the naive predictors, the models were evaluated for the first time. It was decided to make a grid search for the six deep learning models, which were introduced in Section 3.3, for the first model comparison. While designing the LSTMs models the hyperparameter combinations have been tuned only so far, to establish working models for the main experiments.

A grid search applied to set of hyperparameters results in one individual run for every parameter combination possible. This set contains a list of values for every adaptable parameter of the model. If a parameter is considered as reliable and is not investigated any further, the list of that parameter contains a single value. Whereas, when the list of the parameter contains multiple values, each value of this list will be evaluated such that the better parameter can be chosen. This results in as many runs, as there are parameter combinations.

Already, there are six models and three different datasets to be evaluated for every run. Thus, when covering all models and datasets, this leads to $6 * 3 = 18$ *runs* for every combination of the grid search. The number of combinations *combs* of one grid search this leads to a total number of $runs * combs$ runs.

The models used are the MLP models explained in Section 3.3.2. The Simple Multilayer Processor Model (SMLP) refers to the model that also receives the time series data as input, just like the LSTM models, and the Calendar Multilayer Processor Model (CMLP) refers to the lightweight MLP version of AIM that uses calendar information only, as input. Furthermore, the LSTM models used are the MLSTM, the SS2S,the LSTMD and the S2SD. This terminology was introduced in Section 3.3.5. By default the MLSTM and the LSTMD used 2 internal layers and the S2SD and the LSTMD 200 artificial neurons in the cell state.

Additionally, the metrics values of the evaluation runs were retrieved by taking the last validation value calculated for every run. As long as the model did not start to overfit the data, this value presented a reliable estimate of the performance of the run.

Model	Hourly		Daily		Weekly	
	MAAPES	MSE	MAAPES	MSE	MAAPES	MSE
MLSTM	**0.0234**	**0.0029**	0.0485	0.0118	0.0572	0.0176
LSTMD	0.0262	0.0032	0.0441	0.0103	0.0580	0.0186
SS2S	0.0460	0.0072	0.0571	0.0147	0.0562	0.0172
S2SD	0.0276	0.0035	**0.0412**	**0.0092**	0.0624	0.0201
SMLP	0.0398	0.0046	0.0565	0.0137	0.0546	0.0164
CMLP	0.0414	0.0049	0.0546	0.0130	**0.0435**	**0.0101**
Naive Previous	*0.0818*	0.0171	0.0683	0.0239	0.0691	0.0258
Naive Mean	0.0792	*0.0189*	*0.3208*	*0.3233*	*0.3349*	*0.3665*

Table 2: Overview of the model performance of the first grid search with number of epochs set to 1000. In bold printed numbers represent the best value and in italic printed numbers represent the worst value of the respective column.

4.3 Discussion of the First Evaluation

In the first grid search, two parameters were chosen to have different options. This led to $18 * 2 * 3 = 108$ individual runs, since the dropout rate was chosen from the 2 values, 0 and 0.1, and the epoch values were chosen from the three values, 40, 80 and 1000.

The dropout was modified because the MLP of AIM was implemented using the dropout rate 0.1. But it was unclear whether dropout would work for the recurrent models. Furthermore, the number of total epochs was varied, to understand how fast the learning of the models saturates. Saturation of ANNs is a time period where the validation curve does not decrease anymore over time.

For the first two values relative low epoch numbers were chosen because the original MLP at AIM has been trained in a similar range. A relative high value was also chosen as the third value to analyse how the performance would behave in the long run.

The overview of the first evaluation presented in Table 2 shows only two metric values, for reasons of readability: the MSE was chosen in addition to the MAAPES, because the MSE punishes single values deviations more strongly in contrast to the other metrics. The MAAPES focuses more how well a generally is predicted and enables a scale independent evaluation, that is comparable across the datasets.

The first evaluation revealed that most models required a high epoch configuration. Only for the MLP models used with the hourly dataset, a

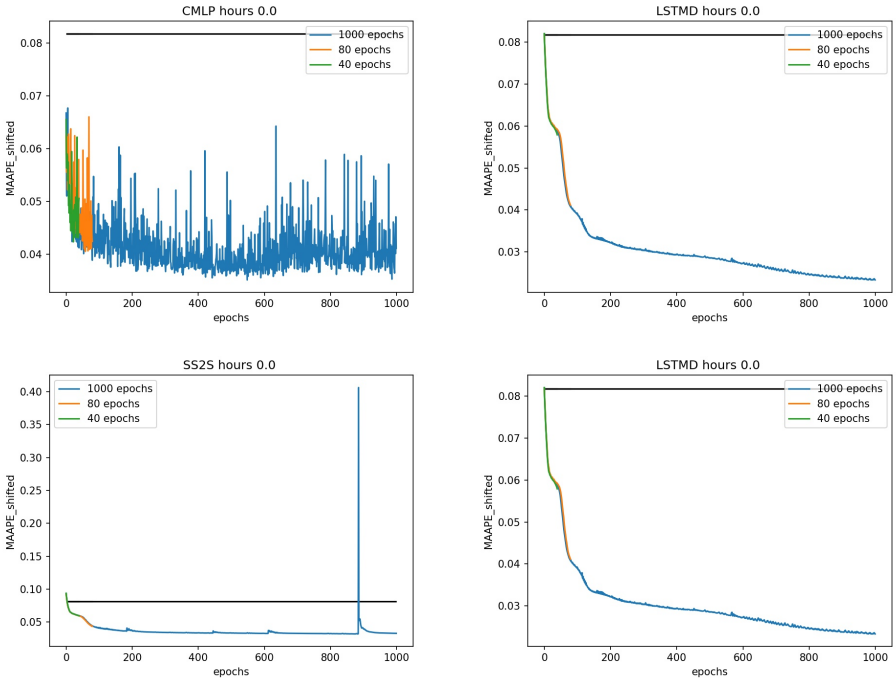

Figure 6: Curves of the validation error of the first grid search, according to the MAAPES: The title shows the model type, the dataset, and the dropout rate chosen. The black bar represents the performance of the best naive predictor and the colored curves refer to different epoch configurations.

saturation was almost reached at epoch 80, see Figure 6. Both MLP models have a more unstable error with a lot of fluctuation, as exemplified in the top left of the figure. Instead, the curves of the other models have a more stable error over the course of the epochs. Furthermore, the SS2S is prone of having artifacts with the error jumping up for a single validation in the error curve, as it can be seen in the bottom left of the figure.

Therefore, in Table 2 only the highest epoch configuration of 1000 is considered and then the mean value over the dropout configurations is taken. The mean of the dropout was taken, because there was no clear winner of the two values for all categories. It can be seen that for the different datasets, different models seems to be better suited.

For the weekly dataset the CMLP seems to perform best with a MAAPES of around 0.044. In comparison, the LSTM based models do not achieve better results than the previous-value predictor for the weekly dataset, neither

does the SMLP by a small margin.

For the daily dataset the S2SD achieves the best values with a MAAPES of 0.041 and for the hourly dataset the MLSTM shows the best performance with a MAAPES of 0.023. It has to be said, that for the more fine-grained measured datasets, generally better MAAPES could be achieved, even for the naive predictors. In the following section the use of dropout for the LSTM will be analysed.

4.4 Analysis of Dropout Usage

As mentioned in Section 4.3, there was no clear best choice for the value of the dropout rate in the first grid search. This is why the dropout rate will be analysed further in this section.

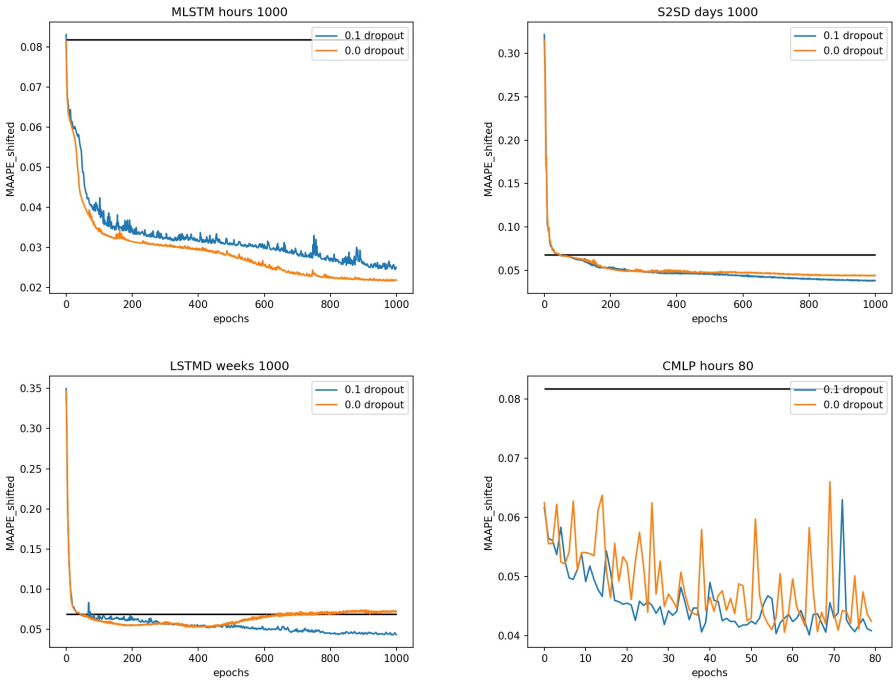

Figure 7: Curves of the validation error comparing the dropout rate, according to the MAAPES: The title shows the model type, the dataset and the number of epochs chosen. The black bar represents the performance of the best naive predictor and the colored curves refer to different dropout configurations.

Setup	No. of MLSTM Cell Layers	S2SD Cell Size	No. of LSTMD Cell Layers & Cell Size
1	2	100	(3, 100)
2	3	200	(4, 100)
3	4	300	(2, 200)
4	5	400	(3, 200)

Table 3: Four adaptations were made to every LSTM based model.

A deeper analysis of the performance of the models used with dropout and without dropout shows that its usage depends on the model and data type. Most LSTM based models prefered not to use dropout, as it is shown at the top left of Figure 7. The only LSTM based model where a dropout rate decreased the error is the SS2S as it can be seen on the top left of the figure.

For a few models with many parameters, like the S2S with the dense layer, the S2SD, or the LSTMD, the error went up again in the region of 200 to 500 epochs only on the weekly dataset. This was interpreted as signs for overfitting, since the weekly training-validation dataset also is relatively small with only around 130 data points and a training on small datasets is more prone to overfitting, in general (Bishop, 2006).

Additionally, the MLP models did prefer dropout usage, but only by a small margin, as shown on the bottom right of the figure. Therefore, it was decided that for the subsequent experiments dropout will only be used for MLPs and the S2SD and for every model when the weekly dataset is used.

4.5 Further Analysis of the LSTMs

This section is about a deeper analysis of the LSTM based models. The MLSTM was designed with the architectural feature to be used with more than two layers, thus the number of layers of it will be adapted. In addition, the number of cell units of the S2SD could be modified and for the LSTMD both adaptations could be made. Therefore, another grid search was created with four adaptable parameters for the three models. The exact adaptations made are listed in Table 3.

Model	Setup 1	Setup 2	Setup 3	Setup 4
MLSTM	0.0426	0.0483	**0.0412**	*0.0576*
S2SD	*0.0456*	0.0452	0.0443	**0.0417**
LSTMD	*0.0438*	*0.0438*	**0.0431**	0.0437

Table 4: Overview of the LSTM fine tuning MAAPES after averaging over all datasets the setup ran on; in bold printed numbers represent the best value and in italic printed numbers represent the worst value of the respective row.

Therefore, another grid search was generated. Because only three models are used this time for the three datasets and the four setups, there were $3*3*4 = 36$ runs to evaluate. Every run was executed with an extended epoch number of 1500. As it can be seen in Figure 8, the models did not completely saturate. The evaluation showed that the different setups helped tuning the model. The differences were relatively small, but still improvements were noticeable. Table 4 shows the averaged values over the datasets for every setup of a model.

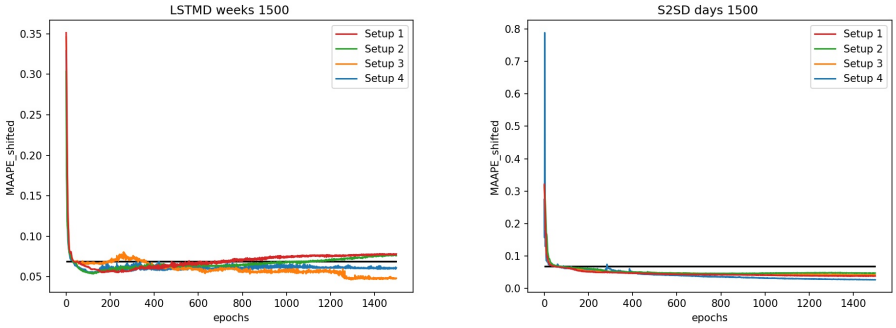

Figure 8: Curves of the validation error of the setup comparison, according to the MAAPES: The title shows the model type, the dataset, and the number of epochs chosen. The black bar represents the performance of the best naive predictor and the colored curves refer to different LSTM setups.

The MLSTM performed best when using four layers of internal LSTM cells. The S2SD showed the best result for 400 cell state units. The table indicates that it does perform better, the more hidden units it has. However, the training time and computation load also increased significantly with a larger cell state. The LSTMD showed the least fluctuation in its validation

error. The best setup, by a small margin, was the setup using two LSTM cell layers and a hidden state size of 200.

The setups that showed the best results, were applied for the final test. This test will be covered in the following section.

4.6 Test Performance

Model	**Hourly**		**Daily**		**Weekly**	
	MAAPES	MSE	MAAPES	MSE	MAAPES	MSE
MLSTM	0.0339	0.0052	0.0623	0.0163	*0.1435*	*0.0962*
SS2S	0.0372	0.0051	0.0741	0.0236	0.0917	0.0445
S2SD	0.036	0.0048	**0.0518**	**0.0107**	0.1249	0.0752
LSTMD	**0.0336**	**0.0041**	0.0613	0.0152	0.1431	0.0929
SMLP	0.0477	0.0065	*0.0887*	*0.0387*	**0.0910**	**0.0394**
CMLP	0.0411	0.0055	0.0637	0.018	0.1381	0.0832
Naive Previous	*0.0891*	*0.0199*	0.0703	0.0252	0.1148	0.0763

Table 5: Overview of the model performance of the final test; in bold printed numbers represent the best value and in italic printed numbers represent the worst value of the respective column.

In this section, the final test performance is presented and it will be discussed. At the end of this section the actual prediction curves will be analyzed.

The final results showed the best results, were the parameter configurations that showed the best performance in the previous experiments. Therefore, for the MLP and the S2SD models and for the weekly dataset dropout was used. Additionally, all models were trained for 1000 epochs and the prefered setups of Section 4.5 were chosen. 16 training runs were generated and the naive previous predictor was calculated for every test set.

The results of this test can be seen in Table 5. First, a comparison to the results of Table 2 shows that every error value increased. Even the error of the naive predictor went up. One the one hand, this might be an indication that the patterns in the test set are generally more difficult to predict than patterns of the validation set the validation set. On the other hand, the test error usually is higher than the validation error, according to Bishop (2006).

The weekly dataset was best predicted by the SMLP, while the SS2S achieved the second lowest test error, both in the range of 0.09. The remaining models did not perform very well and achieved a higher error than the

naive predictor. The CMLP could not achieve the same results as in the previous experiments.

This analysis indicates that the LSTM models actually perform better than the MLP models for some categories of the dataset. It might be that the MLP models in general had the disadvantage of being optimized for another training pipeline, that uses different preprocessing methods and is structured differently. In contrast, the LSTM models might have had the disadvantage of being less optimized in general, in comparison to the MLP models. Because the LSTM models were developed and optimized in the time span of this work only, while a version of the MLP model has already been analysed and optimized for over a year at AIM.

4.7 Analysis of the Prediction Curves

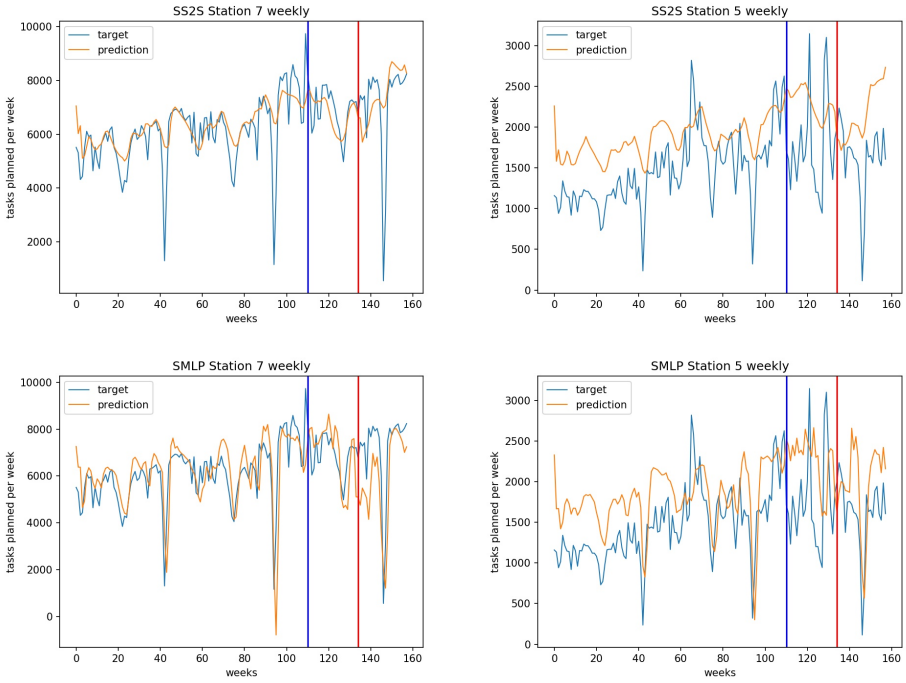

Figure 9: Prediction Curves of the best performing models for the weekly dataset compared: The title shows the model type and the station predicted. The blue line is the position where the training-validation split is located and the red line shows where the test split begins.

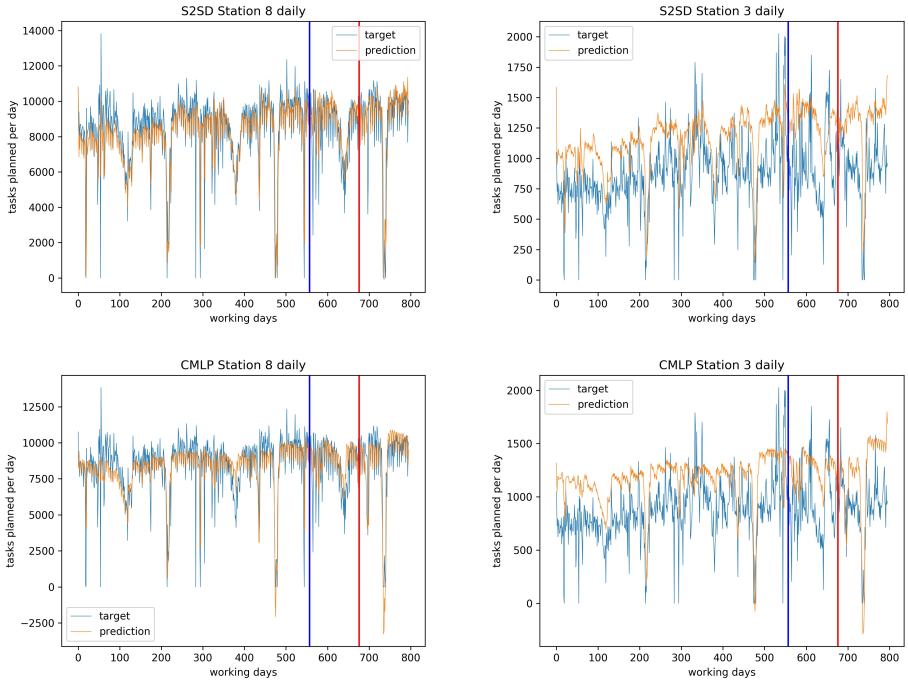

Figure 10: Prediction Curves of the best performing models for the daily dataset compared: The title shows the model type and the station predicted. The blue line is the position where the training-validation split is located and the red line shows where the test split begins.

In Figure 9, the final training curves are presented for the best evaluated model of the weekly category category and in Figure 10 of the daily category. The hourly category is not shown, because it contains to many datapoints to allow a visual analysis of the presented curve in the same manner. For every dataset category four prediction curves were chosen from the ten stations predicted. The left side of both figures shows the predicted stations that were considered the closest to the actual target curve of the two models. For the curves on the right side of the figures, the station was chosen that was considered to be the most difficult station to be predicted by the ANNs. Furthermore, the upper two curves cover the best performing LSTM model and the lower curves the best performing MLP model.

In general, there are certain stations that all models predict relatively well, like station 7 and 8, and other stations, like station 3 and 5, that the models predicts relatively bad. These preferences are even shared across the dataset categories. Additionally, the models tend to develop a similar pre-

diction curve that fits to all stations. Only small adaptations were observed for the predictions of different stations of a model.

This chapter analysed the prediction performance and behaviour of the given models. In the following chapter a conclusion of this work will be formulated.

5 Conclusion and Outlook

All in all, the experiments and implementations can be considered successful. This work indicates that the initial hypothesis seems to be true: in comparison to the MLP models, prediction accuracy could be enhanced when using LSTM based models for time series prediction. It has been shown that the LSTMs performed better when applied to the larger hourly and daily datasets, in comparison to the MLP models. However, the LSTM failed to show a better performance when predicting the weekly dataset.

Additionally, this work provides the necessary theoretical background to aid an understanding of every aspect of the LSTM training process for time series forecasting. Several LSTM architectures have been designed, explained and compared. This might be used as a resource for further applications using LSTM time series prediction.

For further analysis, an interesting approach to the validation of time series models is the application of a time series cross-validation as suggested by Saxe, McClelland and Ganguli (2013). This validation method provides more reliable validation error, since it measures multiple training-validation splits over the course of one training session. Another proposal would be the use of early stopping when deploying the models (Caruana, Lawrence and Giles, 2001). This would help to avoid the impact of overfitting.

Additionally, distinct model configurations could be used for each category of the dataset. It might be that the model comparison differs from this comparison when the models are optimized for a specific data set only. Also, it would be interesting to analyse how the models might behave when given different input and output structures: for example predicting only one station in one session, instead of all ten simultaneously. As has been demonstrated, the models predict certain stations well and others badly. Additionally, more hyper parameters could be modified and tested, and more models could have been developed.

Computing the experiments for this work was a challenging task. This is mainly due to two points: firstly, the models demanded a relatively heavy computational load, so devices belonging to the Institute of Cognitive Science, Osnabrück, were used. There was only limited access to the devices with eight gigabytes of graphics memory, which have been used with a GPU supported Keras version through CUDA[4]. Even so, a single run still took up to 15 hours. Therefore, and due to the limited access to these high performance graphic cards in this work, networks could only be trained for a

[4]https://developer.nvidia.com/cuda-zone

limited number of training epochs. There were also models with many parameters, that needed more memory than eight gigabytes of graphics memory and could not be deployed.

Secondly, the number of runs that needed to be executed for a single parameter configuration was already quite high. Therefore, it was not possible in this context to deploy more and longer experiments. Additionally, the resulting values would be more reliable, if instead of single runs, multiple runs of the same configuration could have been taken and the average results calculated.

A subset of possible models were tested and evaluated. Future projects can continue this work, by testing other model approaches and using the same evaluation process.

6 Bibliography

Bianchi, Filippo Maria et al. (2017). "An overview and comparative analysis of Recurrent Neural Networks for Short Term Load Forecasting". In: pp. 1–41. DOI: 10.1007/978-3-319-70338-1. arXiv: 1705.04378. URL: http://arxiv.org/abs/1705.04378%7B%5C%%7D0Ahttp://dx.doi.org/10.1007/978-3-319-70338-1.

Bishop, Christopher M (2006). *Pattern recognition and machine learning*. Information science and statistics. New York, NY: Springer. URL: http://cds.cern.ch/record/998831.

Caruana, Rich, Steve Lawrence and C Lee Giles (2001). "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping". In: *Advances in neural information processing systems*, pp. 402–408.

Chen, Tianqi and Carlos Guestrin (2016). "XGBoost". In: *KDD '16 Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794. DOI: 10.1145/2939672.2939785. URL: http://dx.doi.org/10.1145/2939672.2939785.

Cho, Kyunghyun et al. (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 1724–1734. ISBN: 9781937284961. DOI: 10.3115/v1/D14-1179. URL: http://aclweb.org/anthology/D14-1179.

Chollet, François et al. (2015). *Keras*. URL: https://keras.io.

Clevert, Djork-Arné, Thomas Unterthiner and Sepp Hochreiter (Nov. 2015). "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: arXiv: 1511.07289. URL: http://arxiv.org/abs/1511.07289.

Ghosh, Shalini et al. (Feb. 2016). "Contextual LSTM (CLSTM) models for Large scale NLP tasks". In: arXiv: 1602.06291. URL: http://arxiv.org/abs/1602.06291.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning*. MIT Press.

He, Kaiming et al. (Dec. 2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 2016-Decem, pp. 770–778. ISBN: 9781467388504. DOI: 10.1109/CVPR.2016.90. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. ISSN: 08997667. DOI: 10.1162/neco.1997.9.8.1735.

Kim, Lark Sang (1993). "Initializing weights to a hidden layer of a multilayer neural network by linear programming". In: *Proceedings of the International Joint Conference on Neural Networks*. Vol. 2, pp. 1701–1704. ISBN: 0780314212. URL: http://www.iro.umontreal..

Kim, Sungil and Heeyoung Kim (2016). "A new metric of absolute percentage error for intermittent demand forecasts". In: *International Journal of Forecasting* 32.3, pp. 669–679. ISSN: 01692070. DOI: 10.1016/j.ijforecast.2015.12.003. URL: http://dx.doi.org/10.1016/j.ijforecast.2015.12.003.

Kingma, Diederik P. and Jimmy Ba (Dec. 2014). "Adam: A Method for Stochastic Optimization". In: arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

Myttenaere, Arnaud De et al. (2015). "Using the Mean Absolute Percentage Error for Regression Models To cite this version :" in: *ESANN 2015 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 22-24 April 2015, i6doc.com pub* I.April, pp. 22–24. arXiv: 1506.04176. URL: http://arxiv.org/abs/1506.04176.

Patterson, Josh and Adam Gibson (2017). *Deep learning : a practitioner's approach*, p. 507. ISBN: 9781491924570. URL: https://www.oreilly.com/library/view/deep-learning/9781491924570/.

Ruder, Sebastian (Sept. 2016). "An overview of gradient descent optimization algorithms". In: arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

Rumelhart, David E, Geoffrey E Hinton, Ronald J Williams et al. (1988). "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3, p. 1.

"Mean Absolute Error" (2010a). In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I Webb. Boston, MA: Springer US, p. 652. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_525. URL: https://doi.org/10.1007/978-0-387-30164-8%7B%5C_%7D525.

"Mean Squared Error" (2010b). In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I Webb. Boston, MA: Springer US, p. 653. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_528. URL: https://doi.org/10.1007/978-0-387-30164-8%7B%5C_%7D528.

Saxe, Andrew M., James L. McClelland and Surya Ganguli (Dec. 2013). "Exact solutions to the nonlinear dynamics of learning in deep linear

neural networks". In: arXiv: 1312.6120. URL: http://arxiv.org/abs/1312.6120.

Summerville, Adam, Michael Cook and Ben Steenhuisen (Sept. 2016). "Draft-Analysis of the ancients: Predicting draft picks in DotA 2 using machine learning". In: *AAAI Workshop - Technical Report*. Vol. WS-16-21 -, pp. 100–106. ISBN: 9781577357735. URL: https://www.aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/viewPaper/14075.

Sutskever, Ilya, Oriol Vinyals and Quoc V. Le (Sept. 2014). "Sequence to sequence learning with neural networks". In: *Advances in Neural Information Processing Systems*. Vol. 4. January, pp. 3104–3112. arXiv: 1409.3215. URL: http://arxiv.org/abs/1409.3215.

Szegedy, Christian et al. (Feb. 2015). "Inception v4". In: *Population Health Management* 18.3, pp. 186–191. ISSN: 1942-7891. DOI: 10.1089/pop.2014.0089. arXiv: 1409.4842v1. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/viewPaper/14806%20http://online.liebertpub.com/doi/10.1089/pop.2014.0089.

Vinyals, Oriol et al. (Aug. 2017). "StarCraft II: A New Challenge for Reinforcement Learning". In: arXiv: 1708.04782. URL: http://arxiv.org/abs/1708.04782.

Yi Jin, Joni Chuang (Synced) (2017). *Tree Boosting With XGBoost — Why Does XGBoost Win "Every" Machine Learning Competition?* URL: https://medium.com/syncedreview/tree-boosting-with-xgboost-why-does-xgboost-win-every-machine-learning-competition-ca8034c0b283 (visited on 02/08/2019).