Paderborn University

Master's thesis

Machine Learning-Based Load Prediction for Network Function Virtualization Scenarios

Christoph Kaiser

Mat.:6701299
kaiserc1@mail.uni-paderborn.de

supervised by
Prof. Dr. Holger Karl
Prof. Dr.-Ing. Falko Dressler

November 26, 2019

Contents

1 Introduction **2**

2 Thesis Background & Related Work **4**

2.1 Virtual Network Function Placement 4

2.2 Traffic Prediction 4

2.3 Related Topics 6

3 Problem Statement and Model **8**

3.1 Network 8

3.2 Services 9

3.3 Traffic 9

4 Available Traffic Datasets **11**

4.1 Feature Description 11

4.2 Choosing a Dataset 13

4.3 Modifications of the Dataset 15

5 Prediction Models **16**

5.1 Long Short-Term Memory 16

5.1.1 Theory & Background 16

Neural Networks 16

Recurrent Neural Networks 17

Long Short-Term Memory 18

5.1.2 Parameters and Architecture of LSTM Models 21

5.1.3 Training an LSTM Model 22

5.2 Auto-Regressive Integrated Moving Average 26

5.2.1 ARIMA Background 26

5.2.2 Identifying the Order of ARIMA and its Parameters . . . 27

5.3 Model Setup and Implementation 28

5.3.1 Libraries and Frameworks 28

5.3.2 LSTM Implementation 30

5.3.3 PmdARIMA (Automatic ARIMA parameter optimization) 32

6 Evaluation **34**
6.1 Test Data . . . 34
6.2 The LSTM Model . . . 36
6.2.1 Identifying the best Architecture and Parameters . . . 36
6.2.2 Training the LSTM . . . 38
6.2.3 Hyperparameter-Tuning Results . . . 38
Hidden Nodes . . . 38
Previous Values . . . 39
Number of Batches . . . 40
LSTM Layers . . . 41
6.3 The ARIMA Model . . . 42
6.4 Comparing LSTM and ARIMA . . . 44
6.5 Optimizing LSTM Prediction . . . 47
6.6 One-Step Prediction vs Multi-Step Prediction . . . 49

7 Future Work and Conclusion **54**
7.1 Future Work . . . 54
7.2 Conclusion . . . 55

8 Appendix **56**
8.1 Documents and Code . . . 56
8.2 Libraries and Frameworks . . . 57
8.3 Figures . . . 57

Abstract **- This master's thesis has the topic of predicting service traffic in a Network Function Virtualization scenario. So, predicting multiple different traffic patterns at each ingress node in a network structure. A problem with current algorithms for placing Virtual Network Functions in a network and scaling the services is that they can typically only react to arriving requests. Replacing this reactive nature with prediction capabilities could yield results for compensating startup times, shutting down services earlier and keep services running. Furthermore, the algorithm can scale the service proactively. Developing a machine learning model that can reliably predict new requests and what services are called by these requests is the goal of this master thesis. A big challenge of this thesis is the modeling of the available information as an input and the output for the neural networks. The second big task is the modeling and training of the machine learning models. The results of the machine learning model are tested against an Auto-Regressive Integrated Moving Average model, as an already established prediction method. These tests show that the developed Long Short-Term Memory is not much better than an ARIMA model. However, both bring their own advantages and disadvantages.**

1 Introduction

Networks with Virtual Network Functions (VNF) are a heavily researched topic. In today's world, the standard way of how a network is constructed is that many network functions (e.g. firewall) have proprietary hardware. As described by [1]:

> A given service usually has a strong connection with some specific middle-boxes. For example, launching a new service needs to deploy a variety of middle-boxes and to accommodate these middle-boxes is becoming more and more difficult.

The difficulty stems from the fact that networks get larger and larger and just adding new hardware becomes more difficult. So a way of using generic server hardware to deploy multiple software solutions would be desirable. As noted by [1], this would lead to lower cost, higher ease of updating functions (e.g. IPv4 vs IPv6) and better maintainability. This is what VNFs are for; they are meant to separate the function from the hardware. That is why it is called Network Function Virtualization (NFV); the services are virtualized and run on generic hardware. NFV then combines these functions to specific services that are provided by a network, e.g. video streaming. Each service is a connected graph of different VNFs. By creating these services, it creates the possibility to scale services if more instances of functions are needed and the NFV scaling algorithm can place the VNFs at any physical node in the network. The scaling of the services makes the network more robust against changing network loads, also saves many resources when a service is not needed and needs less expensive proprietary hardware. However, one of the research subjects of NFV is the scaling of the system and where to place VNFs efficiently. This scaling is determined by the traffic of each service, the more traffic a service generates the more individual instances of the services are started. Furthermore, at what location in the network the traffic passes through determines where the algorithm places the instances. The goal of this master's thesis is to predict the network traffic that requires the algorithm to scale the network and place new VNFs.

However, a problem with VNFs is that they take some time to start. As noted by [2, p.108-109]:

> "[...] even for a single VM, the time needed for boot and basic provisioning is in the order of tens of seconds. This time grows very quickly if a higher number of VMs is required."

When a system can only react to a request and not predict it, and the system has not yet started the VNF, the time for starting the necessary software will be added to the waiting time of the request. This is the worst-case scenario. In a typical scenario, the VNF instance is already running but is overloaded. This overloading of the VNF means the request would enter a queue of this instance and wait for its processing. A request in this context is meant to be a request for a specific service, and all further mentions of request mean the same thing. The approach this thesis takes to handle new requests better is the prediction of those requests. When a system can predict these events, it can start the service and corresponding VNFs early to compensate for its startup time. Also, it can compute in advance where the best location would be to place the VNFs.

An algorithm that handles the scaling and placement of VNFs is [3]. The idea of this master's thesis is to rectify the problems of the reactive nature of these algorithms and provide them with accurate predictions of future traffic. For this purpose, the goal is to develop and test a machine learning model that can reliably predict the incoming traffic of the service requests produced at each ingress node. With these predictions, an algorithm could then not only compensate for the startup time but also shut down services earlier if no new request is arriving or keep services running if a new request is imminent to arrive. The same is true for the scaling of the services. The model will be a Long Short-Term Memory model and, as a comparison, I will develop an Auto-Regressive Integrated Moving Average model. ARIMA models are already established for time-series prediction and are used as a baseline for the quality of the prediction.

2 Thesis Background & Related Work

The next section describes the background on Virtual Network Function placement and also describes some related works. All the related works contain either topics that are also contained in the master's thesis or have the same topic: predicting network traffic.

2.1 Virtual Network Function Placement

After applying Network Function Virtualization and the function is separated from the hardware, the question becomes: how are the VNFs distributed in the network. When a function is no longer bound to specific hardware, it can be placed anywhere in the network, where the necessary compute resources exist. A system can place a VNF wherever it is needed; for example, a firewall VNF can be started anywhere at the edge of the network as close as possible to the entry point of a request, to minimize the route the request has to travel in the network. The possibility of VNF placement then creates multiple problems: how many VNFs are running, where are they placed, when are they shut down and so forth. Algorithms like [3], [4] or [5] try to tackle these problems. Special algorithms are needed because the complexity of the problems increases with a bigger network. Combined with the fact that network traffic can change in a fraction of a second, it is not possible anymore to start and stop VNFs manually. Now even an algorithm that efficiently places and scales VNFs can only do so by reacting to requests. The input of the algorithm would change to the predicted traffic instead of the current traffic. That would lead to the algorithm performing the scaling and placing on future scenarios. This preemptive scaling and placing would lead to the improvement that all VNFs are already ready when they are needed.

2.2 Traffic Prediction

The topic of predicting network traffic using deep machine learning is a relatively new one, nonetheless there exist a lot of different papers on the topic. Using models like the Auto-Regressive Integrated Moving Average models to predict network traffic is an older topic, but multiple of the related works

show that the machine learning approach can outperform the ARIMA model. An example of using Long Short-Term Memory models for predicting network traffic is [6]. In this work, LSTMs are compared to a model using a combination of a Convolutional Neural Network and an LSTM. The results show that a machine learning model can accurately predict network traffic even when the input data is quite sparse. Of course, the prediction is not perfect but a close approximation of the reality. The topic of the paper is similar to the topic of this thesis, but this master's thesis goal includes the prediction of not only one straightforward traffic pattern but multiple real-world network traffic recordings.

R. Vinayakumar et al. [7] implement multiple machine learning models and compare their performance at predicting network traffic. They implement multiple Recurrent Neural Network models, including LSTMs. According to their results, the LSTM performs the best but with a high computational cost. Other models that perform well are the Gated Recurrent Unit(GRU) and Independent Recurrent Neural Network(IRNN) models, the GRU with less computational cost than the LSTM. They conclude that with more computation resources, they could train more complex networks and improve the results even more. The results enforce that LSTMs can predict network traffic, but it also shows that something to look out for is the complexity of the computations.

[8] is also using a machine learning approach to predicting network traffic, mobile traffic in this case. They are using a dataset that contains the mobile traffic split up into cells and arranged in a grid. Their RNN cells are also of the LSTM kind and they construct multiple different models including Convolutional RNN as seen in a previous paper or a 3D-RNN. With these models, they compare the performance at predicting *call detail record* (CDR) per grid cell. Using the same dataset [9] tried to predict the *control plane traffic*. This paper also uses multiple machine learning approaches and compares the results. The two approaches the authors compared were a basic Neural Network and an LSTM. However, the LSTM performed better with fewer nodes and the same amount of data, so they used the LSTMs for their approach. Both of these papers show that LSTMs can recognize traffic in traffics and is able to predict it.

A paper that does not use deep machine learning is [10]. They used an ARIMA model to predict mobile traffic. The model produces good results for the traffic from the mobile cells. However, looking at the traffic, it has many similarities from one day to the next, but they did use two datasets: one with no daily changes in traffic and one with less traffic at the weekends. Here the question would be how the ARIMA model performs with traffic that changes more from one day to the next, which is the case for this master's thesis. However, this work shows that, in principle, the ARIMA model is well suited for time series prediction. This reinforces my decision to use the ARIMA model as a baseline for this thesis.

2.3 Related Topics

The following scientific works are related to this master thesis. They are not trying to achieve the same thing, predicting network traffic, but parts of these works can be found in this thesis.

The first paper is [11]. It uses ARIMA models to simulate realistic network traffic. They not only simulated periodic traffic but also traffic that is linearly increasing and traffic with anomalies. This paper shows that the ARIMA model used in this master's thesis is a suitable model to simulate and predict network traffic.

Another related paper is [4]. The paper demonstrates a way of optimizing a VNF placement and scaling algorithm with prediction. The paper also describes that it is an online prediction. So the algorithm is learning at run-time and is not trained beforehand. The algorithm from this paper is called *VNF Provisioning for Cost Minimization* and the online learning method they are using is called FTRL(Follow-the-Regularized-Leader). No machine learning model from this thesis appears in the paper, but the paper shows nonetheless that predicting network traffic can lead to an increase in quality of the VNF placement and scaling.

The authors present a different approach to placing VNFs in [12]. With this method, a particular model, model predictive control, is used, that tries to maintain a given output value. It makes a prediction and takes the new output to adjust its forecast for the next time-step. Again this paper demonstrated

how predicting network traffic could improve the performance of placing VNFs.

Also, a different use of machine learning in the realm of NFV and VNFs is the prediction of resource usage of VNFs. That is what [13] tries to do. Machine learning is applied to predict the CPU usage of VNFs with just the network traffic as an input. This paper proves that a machine learning model can find patterns in network traffic that it can reliably predict, just as this master's thesis tries to do.

3 Problem Statement and Model

This section further elaborates on the problem and how it is modelled.

3.1 Network

The actual problem domain is a network modelled as a directed graph $N = (V, L)$, with no limits to the number of existing nodes. In reality are the nodes contained in V some kind of hardware on that virtual network functions can be started. The links L represent the physical connections between these hardware elements. Figure 1 shows a small example of such a network. The network can also have any number of ingress and egress nodes. In the example, the network has two ingress and one egress node ($A, C, B \in V$). The connections $x_1, x_2, y_1 \in L$ describe the ingress and egress connection of the system. These connections represent an infinite number of possible connections to nodes that request a service or receive the result of a service process, that is why they are substituted as simple arrows in the figure. Each node can be ingress and egress node at the same time, meaning it can have an ingress and egress connection. All the incoming traffic flows over the connections x_n and over the connections y_n, where all outgoing traffic leaves the system.

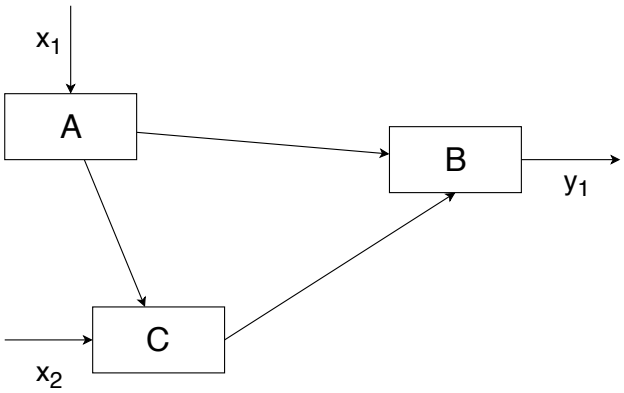


Figure 1: Example for a network with two ingress and one egress node.

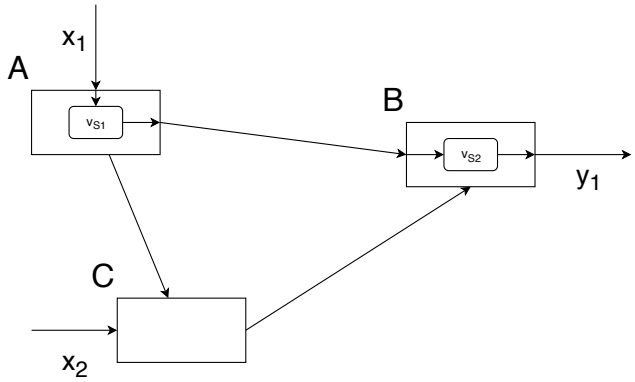


Figure 2: Previous network with an example service included.

3.2 Services

Services are modeled with directed acyclic graphs $S = (V_S, L_S)$. Each node in V_S represents a function (Virtual Network Function), which gives the result to the next node in the graph or outputs it from the network. The function nodes can be placed at any node of the network N. One restriction is the fact that the connection from L_S has to be mapped to the connections L of N. Figure 2 displays an example of this. The service with two VNFs, $v_{S1}, v_{S1} \in V_S$, is placed in the network with v_{S1} placed at A and v_{S2} placed at B. Placing v_{S1} at C and v_{S2} at A would not be possible since there is no path from C to A and the necessary connection of the service could not have been mapped to the network.

3.3 Traffic

Now coming back to the actual problem of predicting the network traffic. The relevant parts for the problem of this thesis are the incoming traffic and the nodes where the traffic arrives. The structure, the internal traffic (between ingress and egress nodes) and the output traffic of the network has no relevance for this problem solution. So the actual feature that this thesis predicts is the traffic for each service at each ingress node. The traffic x_n splits up into the traffic for the different services.

What is also important for the prediction of the traffic apart from the system itself are additional information about the traffic. One of the most important

examples is the date and time of the incoming traffic. Since at different days and time of day the traffic changes drastically.

4 Available Traffic Datasets

There exist many different sets of networking data. All offer different data, from IP-packets to the workload on the hardware in the network. This chapter describes the features of different datasets and which of these sets were chosen to train the models. Datasets that I considered for this thesis:

- Alibaba data center set [1]
- Network simulation data [14]
- 2.5 years of Google DNS traffic [15]
- University IP-Packets annotated with applications [16]
- GÉANT research network traffic [17], [18]
- Abilene research network traffic [17], [18]

4.1 Feature Description

To choose the best set for training the models, analyzing six different sets was also part of this thesis. One of the sets the Alibaba corporations collected on their data centres. This set contains information about the machines included in the network. The data contains the job executions of the single machines and what batches they are currently working on. All of this information is split up into multiple files. Instead of communication between machines, this set represents workload on machines.

Another set is a set provided by [14]. It represents the results from a network simulation and all the network recordings. The data contains features such as the network topology, routing information, bandwidth, transmitted packages, dropped packages. More information can be found on the GitHub page[2].

[1] `https://github.com/alibaba/clusterdata` Accessed: 2019-11-23

[2] `https://github.com/knowledgedefinednetworking/NetworkModelingDatasets/tree/master/datasets_v0` Accessed: 2019-11-09

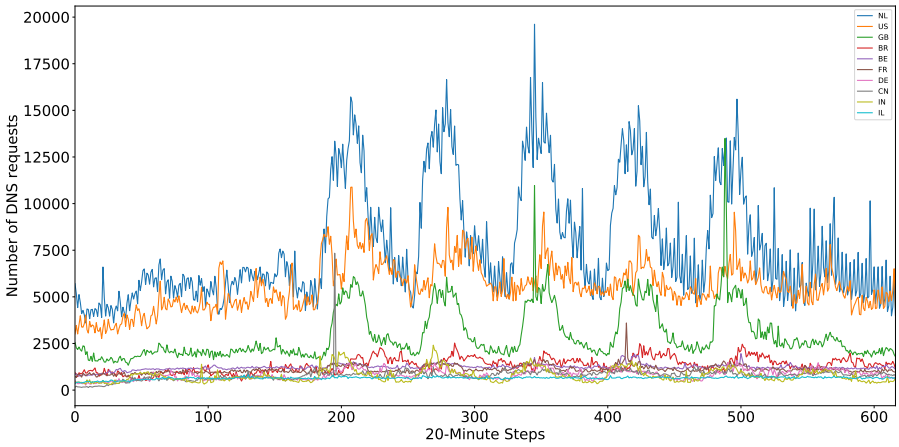


Figure 3: Plot of the data in the Google set. Combined into 20-minute steps displayed are only the first ca. 600 steps with only the 10 countries with the most DNS requests.

A much different set is a set of DNS requests from Google datacenters spanning 2.5 years. [15] gathered the data[3]. Included in the set is every single request from these 2.5 years. The information for each request includes when it arrived, from which country it came and which data-centre processed the request. It is by far the most extensive of the six sets. Figure 3 shows an excerpt of the data. The plot shows the number of DNS requests combined into 20-minute steps displaying the first ca. 600 time-steps. Each line represents a country of origin where the DNS requests are coming from.

A similar dataset is a set from [16]. This data includes IP-packets of a university network, each packet labelled with application information. Examples include Google, HTTP or YouTube. The plot in figure 4 shows how the data changes over time. It also shows a problem with this set. The recording intervals were not adjacent to one another, leading to vast stretches with no data.

A set containing another type of data is the set from [17], [18]. It contains the bandwidth of all pairs communicating in the network, combined into 15-

[3] `https://www.simpleweb.org/wiki/index.php/Traces#Passive_Observations_of_a_Large_DNS_Service:_2.5_Years_in_the_Life_of_Google` Accessed: 2019-11-09

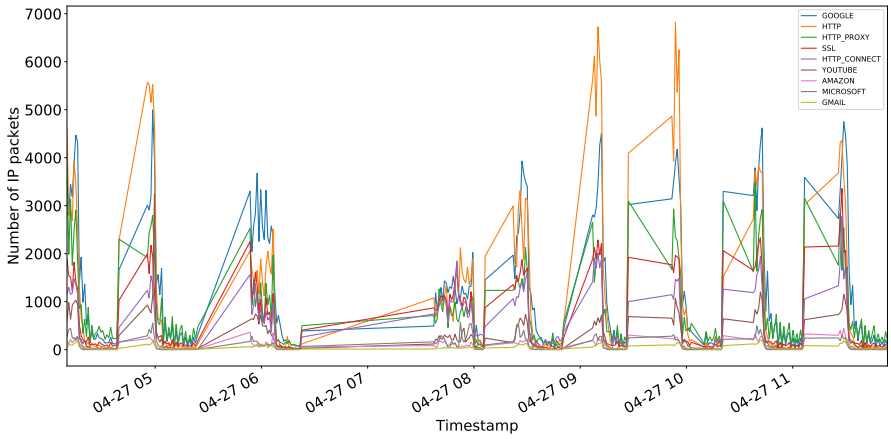


Figure 4: Plot of the set from Kaggle. Included are the nine most requested apps.

minute steps. The data originates from the GÉANT network, a European research network, including 23 nodes all over Europe. The plot 5 shows the data from the first week with the source as node 1 and destinations nodes 1 to 6. The data was anonymized, that is why a number only denotes the nodes. Also, the data was collected over four months.

The last data collection originates from the Abilene network in the USA from [17], [18]. It is in the same format as the previous dataset, but this network includes fewer nodes then the GÉANT network. However, it has a higher accuracy with 5-minute steps and the recording ran over an extended period of 6 months. Here an argument can be made for both datasets, depending on what is needed: more diversity in the nodes or higher accuracy.

4.2 Choosing a Dataset

The goal of this thesis is the prediction of the different service requests for each ingress node. With that in mind, a dataset would be ideal that either provides the service request per ingress node or a dataset that can work as a substitution for service traffic. Since none of the six datasets contains actual service traffic, one of the six has to substitute the service traffic.

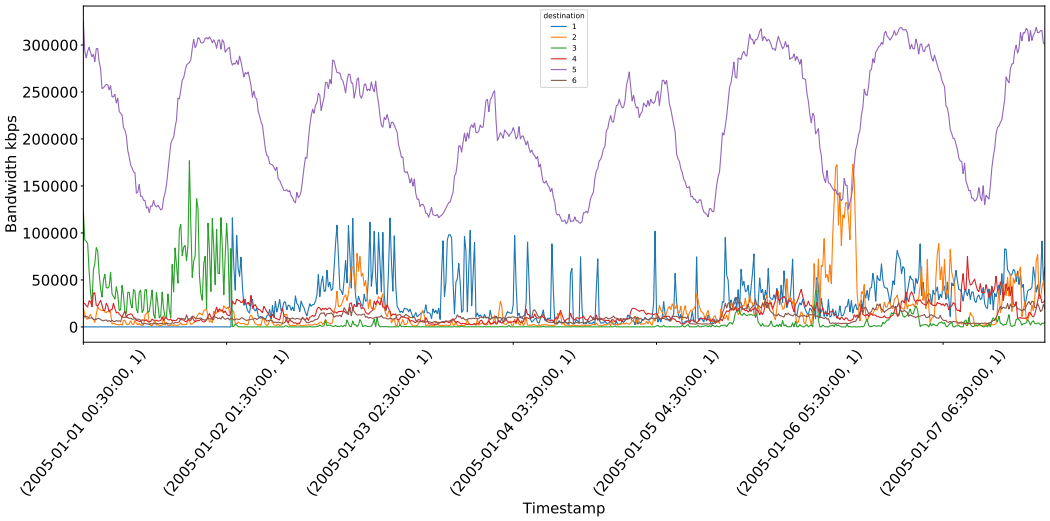


Figure 5: Plot of the network bandwidth from GÉANT network from source 1 to the first 6 destinations.

The set from Alibaba [4] and the collection from the Kaggle platform [16] are both not a good fit for this application. The Alibaba set is not representing the network traffic itself but the jobs and batches running on the machines in the network. Maybe with much work, the traffic could be inferred from the jobs running on the machines, but that is not worth the work for this thesis when other sets are available. The IP packet set from Kaggle, on the other hand, would represent a good substitution for service traffic. With a reinterpretation of the apps as the services, the service traffic would be equal to the app traffic from Figure 4. However, the long periods where data is missing in the traffic make it not suitable for training a machine learning model.

Furthermore, the data from the knowledge defined networking dataset [14] is not the best choice for the work of this thesis. Since the data of this set is the result of a simulation, the credibility of real-world data is missing. Because the developed models should be trained with real-world data to get realistic results, I will not use this data.

The DNS requests in the Google dataset [15] are a better fit at the first look. It is a highly detailed dataset; each DNS request is recorded and fitted with a

[4] `https://github.com/alibaba/clusterdata` Accessed: 2019-11-23

timestamp. However, since these records are DNS requests, it would not be the best idea to develop models that predict network service traffic, since network service traffic is behaving differently than DNS traffic. Also, what is missing are ingress nodes, that would lead to much reinterpretation and modification of the data, which would decrease the credibility of the results.

The two last sets are the GÉANT network data and the Abilene set [17], [18]. Both represent real network traffic between nodes and contain the traffic of all pairs in the network. Both datasets also span many months. So there is much variety to choose from with traffic at weekends, workweek traffic and also traffic at bank holidays. In the end, the choice fell on the GÉANT network data because of the more substantial number of nodes and the resulting diversity.

4.3 Modifications of the Dataset

After choosing the dataset, the task is to interpret and add to the data so that a model can be trained that predicts service network traffic. When looking at the GÉANT network, the data contains all pairs and their bandwidth in 15-minute steps. So for the model, a network $N = (V, L)$ is needed where each service traffic x_n can be defined to each ingress node $I \in V$. This requirement needs to be applied to the GÉANT network data and can be fulfilled when taking a subset of the nodes and looking at the traffic of the pairs. The subset that was taken can be interpreted as the ingress node and the traffic going to these nodes can also be interpreted as the bandwidth for each service. As an example, if the service subset is node 1 to 10, then the remaining nodes 11 to 23 are interpreted as the ingress nodes. This results in a detailed record of how much bandwidth each service to each ingress node uses. The reinterpretation also has the positive effect that I do not have to change the data.

The same assumption can be made for the Abilene set [17], [18]. The only difference here is the number of nodes, that leads to more diversity in the network GÉANT network; that is why I chose the GÉANT network.

5 Prediction Models

This next chapter describes the Long Short-Term Memory and Auto-Regressive Integrated Moving Average model in detail and what the important topics are when building and training models.

5.1 Long Short-Term Memory

The main focus of this thesis, the LSTM, is described in this section in more detail. The description includes the theoretical background to understanding the basics of LSTMs as well as the practical aspects of building and training an LSTM model.

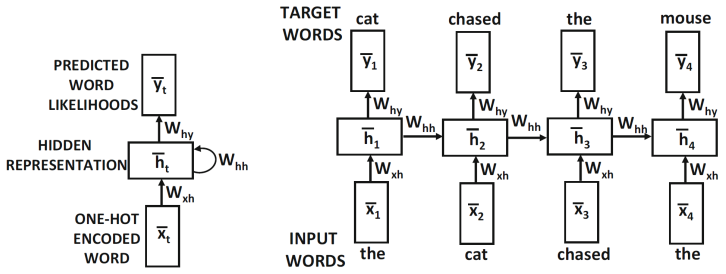
5.1.1 Theory & Background

Neural Networks Neural Networks are a form of machine learning models. The idea behind them is that they mimic a human brain. Every neural network is made up of single neurons in different layers; each neuron connects to each neuron of the next layer. There are three types of neurons: input, output and hidden. Input neurons get the data into the network, output neurons provide the results and hidden neurons modify the data on the way between the input and output. The value of a neuron is the weighted sum of all neurons in the previous layer and for each neuron the weights are different. These weights are the central part that the training is modifying. This weighted sum is then mapped to a function to calculate the output; one example for such a function would be the Sigmoid function [19, p. 11].

A neural network learns by getting fed input data and after starting with random weights and therefore random outputs, the model checks if the output is correct. With a function, it then tries to change the weights of the neurons to get closer to the correct output. After feeding the network with enough data, the function that describes the behaviour of the model should converge on a local minimum. There exist multiple different functions to calculate the weights; one example is gradient descent [20, Ch. 10.5.a]. A loss function judges the quality of the predictions, which the optimization function tries to make as small as possible, finding a local minimum. One loss function that

authors use often is the *Mean Squared Error*. Section 5.1.3 closer elaborates on the training of neural networks.

The previously described behaviour is the essential function of a neural network, but there are multiple versions of neural networks, including Recurrent Neural Networks closer described in the next paragraph.



(a) RNN (b) Time-layered representation of (a)

Figure 6: The Structure of a RNN a), and its time-layered representation b). [19, p. 275]

Recurrent Neural Networks Recurrent Neural Networks (RNN) are also a form of Neural Networks. However, the difference between a standard Neural Network and an RNN is the preservation of previous outputs, as described in [19, p. 274ff.]. As can be seen in Figure 6 a) the RNN has an input $\bar{x}_t$, an output $\bar{y}_t$ and a number of hidden layers $\bar{h}_t$, but what is new in RNN is the self-loop with weights W_{hh}. With this loop, information of previously hidden layers is passed to the next instance of the hidden layers when a new input arrives. This concept gets clearer when viewing Figure 6 b) with every new input the output of the previous $\bar{h}_{t-1}$ gets transferred to the next $\bar{h}_t$. The input $\bar{x}_t$ for the example in 6 b) would be the vector [the, cat, chased, the]. Each of the elements will be an input for a step. So the RNN would start with the input *the* and use that prediction combined with the next input *cat* for the next prediction continuing with the whole input vector. As described in [19, p. 276] the output vector gets combined with the new input:

$$\bar{h}_t = tanh(W_{xh}\bar{x}_t + W_{hh}\bar{h}_{t-1})$$

From this basic version of RNN multiple different more complex variants diverged. I use one of those variants in this thesis; the LSTM(Long Short-Term Memory) model.

Long Short-Term Memory The basics of the LSTM are the same with the RNN: it has an input and output and passes its previous output to the next step. The difference lays in the fact how the LSTM combines the different outputs and inputs and how the model transfers them to the next Neural Network step. Additionally, to the output, a combining cell-state is also transferred in each step. This cell-state holds information covering all the steps that were executed previously. The model updates this state in each step. The difference of the LSTM to a standard RNN can also be seen inside the cells. In a basic form, the LSTM is made up of the following:

- cell-state c_t:
  - forget gate: input is used to decide what to "forget" from the copied cell-state
  - new cell-state: with the input and the modified cell-state the new state of this cell is calculated.
- output h_t:
  - the input is combined with the new cell-state to determine output.

The described structure is the basic idea behind LSTMs. An example of one implementation of an LSTM-cell and how the input, cell-state and output are combined can be seen in Figure 7 with the corresponding formulas explained in the following paragraph.

$$\text{Input Gate: } i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)$$

The input gate determines what information of the input x_t and h_t should be used in the calculation. If a value in the vector i_t is zero the value is not used and if it is one it is used fully.

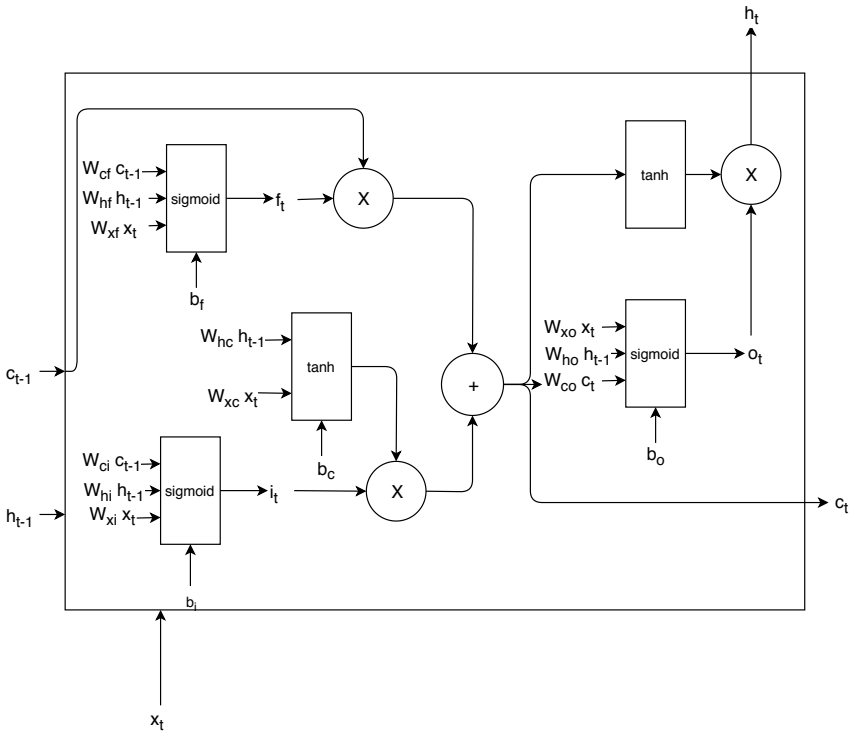


Figure 7: Visual representation of an LSTM cell [21]

Forget Gate: $f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)$

The forget gate functions similar to the input gate but this gate determines what is "forgotten" from the last cell-state. This can be seen in the formula of the new cell-state c_t: $f_t c_{t-1}$.

Cell-state: $c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$

The cell-state is created from the combination of the forget gate f_t with the old cell-state c_{t-1} and the input gate i_t combined with the input x_t and the last output h_{t-1}.

Output Gate: $o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o)$

Output: $h_t = o_t tanh(c_t)$

Finally the output gate modfies the inputs and the cell-state to produce the actual output of this LSTM step. The diagram in Figure 7 represents one cell in an unrolled version of an LSTM, just like in Figure 6 b).

First looking at the inputs in the diagram: c_{t-1}, h_{t-1}, x_t. These inputs represent the cell-state (c_{t-1}) and the output (h_{t-1}) of the previous step and the input (x_t) to this step. c_{t-1} and h_{t-1} are both vectors of a length that corresponds to the number of hidden nodes in the LSTM. The input (x_t) to an LSTM is always a 2D-Array; its shape is (time-steps, features).

Let me make a concrete example to understand the LSTM better. We have a sequence of numbers; [1, 2, 3, 4, 5]. The goal is to predict the 3rd number using the previous two numbers. Then the input array would have a *time-steps* dimension of two, because I have two steps in our sequence (e.g. 1,2) and the *feature* dimension of one, because I have one feature, one time-series. However, for the calculation at step t, only the element t of the input vector is used. So the actual input x_t in this example is a vector of shape 1x1 per step. The input to the first step would be $x_0 = [1]$ and the second step $x_1 = [2]$.

After defining how I shaped the input and output, the form of the weight matrices also becomes apparent. All internal vectors have the same length as the output vector. So i_t, f_t, c_t, o_t, all must have the same length as h_t. All weight matrices W must have the form of $m \times n$ where n is the same dimension as the output vectors and m depends on the vector this matrix is multiplied with. When using the previous example with the number sequence the input x_t is the shape 1×1 the weight matrices W_{xi}, W_{xf}, W_{xo} must have the shape $1 \times n$, where n is the width of the output.

5.1.2 Parameters and Architecture of LSTM Models

This section elaborates closer on the previously described elements of the LSTM and how they can be parameterized and how multiple LSTM layers can be combined. I used Keras for building the models and use it here as an example of an LSTM implementation. In Keras the whole LSTM can be customized: activation functions to use, if a bias should be used or not, if weights should be constrained and other values.[5] Nevertheless, the underlying architecture of the LSTM can not be changed: input-, output- and forget-gate and the cell-state are part of the LSTM. However, when staying with the standard LSTM configuration, since I wanted to test a basic LSTM with it standard functions and values. Then the parameters that one should set are the number of hidden nodes (units) and if the model should return whole result sequence (predictions of each step). The number of units describes the number of hidden nodes and the length of the output vector. As described above, it also defines the width of all other vectors and matrices in the LSTM cell and fundamentally set the complexity of the network. Each LSTM layer can have a different number of hidden nodes. When combining multiple LSTMs, LSTM layers that come before another LSTM layer have to return the whole sequence of outputs as an input for the next LSTM. This concept becomes more evident when looking at Figure 12: a) shows a single LSTM cell in Keras and its input and output dimension. It gets an input of shape (batch size, time-steps, features) but the output is in shape (batch size, units). The output is missing the time-step dimension because only the prediction of the last time-step is

[5] `https://keras.io/layers/recurrent/` Accessed: 2019-11-16

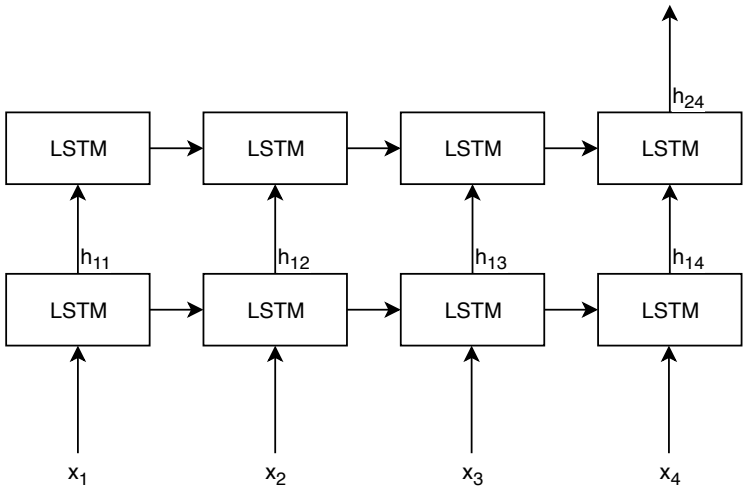


Figure 8: Unrolled version of a two layer LSTM architecture.

returned and not each output of every step. This is different when looking at the two-layer architecture of the LSTM (Figure 12 b)). In this example, after the first layer, all results are returned and put into the next LSTM layer. How the batch size factors into the LSTMs and neural networks in general is explained in the Section 5.1.3.

To better understand this concept, Figure 8 contains an example of an unrolled two-layer LSTM with four time-steps. In this example all outputs $h_{11}, h_{12}, h_{13}, h_{14}$ are returned and put into the second LSTM layer. Moreover, the second layer then only returns the result of h_{24}.

5.1.3 Training an LSTM Model

The basics of training recurrent neural networks are the same as for all neural networks: the goal is to change the weights of the hidden nodes so that the results get closer to a minimum of the loss function. As already said, the idea for recurrent neural networks is the same, but it is a more complex problem since one neural network has to change inputs over multiple steps and feeds information forward. Nevertheless, a close description of how the problem was solved mathematically for recurrent neural networks would be out of scope for this thesis. Because of that, I only give a general description of the ideas behind the optimization of neural networks. For a closer look at recurrent neu-

ral networks, [22] surveys multiple different approaches modified for recurrent networks.

For the description of how to optimize a neural network, the two main factors are the loss function and the optimizer. The primary approach is to start with random weights and change the weights after the first result, so the result gets closer to a local minimum of the loss function. First about the loss function: for this function exist many different approaches, the Keras documentation list multiple approaches[6]. Each of these functions calculates some form of value that describes the quality of a prediction. One example of the more popular functions is the *Mean Squared Error* function. It is made up of the simple formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y_i}\right)^2$$

The Mean Squared Error function is often used to judge the performance of a prediction. Papers like [7] and [8] use the MSE or a different form like the Root Mean Squared Error but do not reason why they are using these functions. However, It is clear that the MSE function is a simple and efficient way of representing the prediction error.

After calculating the loss function, with for example the MSE, the weights need to be changed. For this, an optimizer is used. Here are also many examples in the Keras documentation[7]. These optimizers use different approaches to change the weights. One of the frequently used approaches is gradient descent [7]:

> Gradient descent is one of the most commonly used techniques to minimize the loss function.

As one example for an optimizer is the ADAM optimizer [23], it is also often used in examples for neural networks. ADAM also uses gradient descent. The idea behind gradient descent is simple: the gradient of the function is determined and the gradient is followed down to find some local minimum. When looking at the result of the loss function, the first step is computing the

[6] `https://keras.io/losses/` Accessed: 2019-11-16
[7] `https://keras.io/optimizers/` Accessed: 2019-11-16

gradient of the loss function. On these results, the gradient descent updates the weights so that the result of the loss function moves down the gradient.

Keras calculates the gradient via backpropagation, which is also standard practice in machine learning. Backpropagation is a recursive algorithm to find the steepest gradient of a function. It starts with the output neurons and looks at the error of each output and how much it needs to be increased or decreased to get closer to a local minimum. To determine how the output neuron can be changed, the backpropagation algorithm takes the previous layers and its weights into consideration. When looking at the concrete example in Figure 9, the formula for describing the one output neuron is as follows:

$$a = a_3 \times w_7 + a_4 \times w_8 + b$$

$$a_5 = \begin{cases} 0, & \text{for } a < 0 \\ a, & \text{for } 0 \leq a \leq 1 \\ 1, & \text{for } a > 1 \end{cases}$$

Since the activation of one neuron should not be bigger than 1 or smaller than 0, the function includes the two edge cases. With concrete values the formula looks like this:

$$0.5 = 0.1 \times 0 + 1 \times 0.5 + b$$

The bias b, in this case, is 0. Lets say the actual desired output should have been $a_5 = 0.1$. Three things can be done to decrease the output:

1. decrease bias
2. decrease the weights
3. change the activations of the previous layer

The first step, decreasing the bias, would here mean to change it to a negative value to get closer to the output of $a_5 = 0.1$.

Now looking at the weights: here the values of the previous activations a_4, a_3 are taken into consideration. The neuron a_4 has activation of 1, so it has the maximum value and the other neuron a_3 is at 0.1. Because of the limitation of the calculation to bigger or equal to 0 and smaller or equal to 1, the maximum

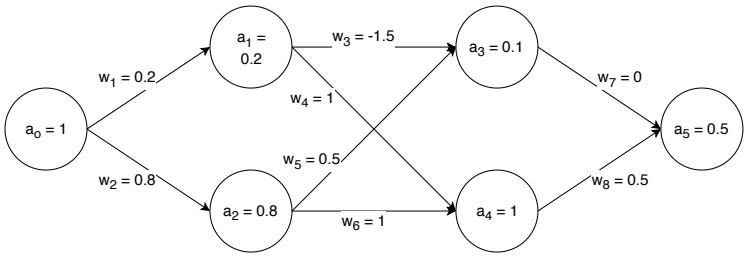


Figure 9: Basic example for a neural network with concrete values and linear activation.

value for a neuron is 1 and the minimum 0. So the weight w_8 will be decreased more than w_7 because the change in the weight w_8 will have more influence on the result of the output because of the higher activation of the neuron a_4.

The third step, the change in the activation of the previous neurons, can only be done indirectly, since the activation of the neurons is calculated from previous activations and weights. How much the neuron activation should change is also in relation to the weights. That means neuron a_4 with weight w_8 will be decreased more than the neuron a_3 with weight w_7, since w_8 has the higher weight and more influence on the output because of that. Now knowing how the activation should be changed, the algorithm does the same for the next layer since the direction of the change in activation a_3 and a_4 are now known.

The backpropagation is a very time-intensive computation to do. Because of that, the batch size mentioned in the previous section, 5.1.1, becomes relevant. Theoretically, in each training step (epoch), the whole data is shown to the network and the backpropagation is performed for each value and gradient descent is then performed on the average of all backpropagations. When one defines a batch size, the data is split up into multiple sets, each the size previously defined. Then the network is only shown one batch per training step (epoch). The result is that the backpropagation and gradient descent is only performed on one batch. This makes the backpropagation and gradient descent less accurate, but the computation time of the backpropagation is reduced drastically.

5.2 Auto-Regressive Integrated Moving Average

The term ARIMA stands for Auto-Regressive Integrated Moving Average; it is a modified version of an ARMA model. It was introduced in 1970 by Box & Jenkins [24]. These models are used to predict or simulate time series, but they are no deep learning models like neural networks. The ARIMA model is also used to predict a time series. In an ARIMA model three things are combined: An autoregressive model (AR), moving average model (MA) and an integral part (I). This section gives a short overview of the basics of ARIMA models.

5.2.1 ARIMA Background

When looking at a standard ARIMA model, one can describe the model through three variables: (p,d,q). As described by [25] the three variables stand for:

- p = order of the autoregressive part
- d = degree of first differencing involved
- q = order of the moving average part

The autoregressive model (AR) part of the ARIMA model is using past values of the predicted variable, to model it. This part is described through the p variable. What exactly the p stands for becomes clear when looking at the formula of an autoregressive model [25]:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + \epsilon_t$$

So p is just the number of previous values to take into account. The current variable y_t is described by previous values of itself. All values are also modified by a parameter ϕ_t. Also included in the formula is a constant c and an error ϵ_t that is just white noise.

When looking at the moving average model (MA), it looks similar to the autoregression model [25]:

$$y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + ... + \theta_q \epsilon_{t-q}$$

But here the variable y_t is described through past forecast errors ϵ, also modified with a parameter θ_t. It is again contained in this formula a constant c and an error ϵ_t which is also white noise.

Now the third part, the differencing, and how it is defined. The formula for the differencing in first (a) and second (b) order are [25]:

$a)$
$y'_t = y_t - y_{t-1}$

$b)$
$y''_t = y'_t - y'_{t-1}$
$= (y_t - y_{t-1}) - (y_{t-1} - y_{t-2})$
$= y_t - 2y_{t-1} + y_{t-2}$

When differenced once the y'_t is basically the change from one value to the next. For the ARIMA model, usually only the first and second-order differencing are used.

Putting all three parts together results in the following model:

$$y'_t = c + \phi_1 y'_{t-1} + ... + \phi_p y'_{t-p} + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} + \epsilon_t$$

In this case y'_t could be differenced more than once. This is the basic version of an ARIMA model, but what ARIMA is also capable of is predicting seasonal data.

When working with seasonal data the ARIMA notation changes to (p,d,q) (P,D,Q)m, with m denoting the number of observations per season. The definition for the (P,Q,D) part is similar to the non-seasonal models, but the shift in the data changes. So for example, if one uses a p of 1, the term would look like this: $\phi_1 y_{t-1}$. For the seasonal data with 4 observations per season (m=4), a P of 1 would look like this: $\Phi_1 y_{t-5}$. Additionally to the backshift of 1, the data is also shifted back a whole season. The seasonal terms then get added to the non-seasonal term; that is the seasonal ARIMA model.

5.2.2 Identifying the Order of ARIMA and its Parameters

The identification of the order of the single parts of the ARIMA model can either be made manual or automatic. Since the ARIMA model is not the

central part of this thesis, the exact procedure and mathematical background is not explained, but I give a basic overview. The basic process is displayed in the flowchart Figure 10. As already said, the ARIMA is not the main part, so the automatic approach was used for this thesis. This is described in the Evaluation Section 6

After the order was selected, the next step is fitting the model to the function. The actual fitting of the ARIMA model is similar to the training of a neural network. The parameter c, ϕ, θ are optimized to get the best solution [25]:

> Once the model order has been identified (i.e., the values of p, d and q), we need to estimate the parameters $c, \phi_1, ..., \phi_p, \theta_1, ..., \theta_q$. When R estimates the ARIMA model, it uses *maximum likelihood estimation* (MLE). This technique finds the values of the parameters which maximize the probability of obtaining the data that we have observed.

More information about identifying the order and parameter optimizations can be found in [26], [27] and [24].

5.3 Model Setup and Implementation

This section closer elaborates on the concrete libraries and frameworks that were used to produce the results of this thesis.

5.3.1 Libraries and Frameworks

For creating the machine learning models, I chose the library Keras[8]. Keras is a library that enables a user to design, train and use deep machine learning models. For that, Keras uses one of multiple different machine learning libraries (TensorFlow, Theano or CNTK), TensorFlow in the case of this thesis.

Algorithm 1 shows the basic construction of a machine learning model in Keras. The first step is to define a sequential model[9], then add as many layers with their configuration (number of hidden nodes, output shape, activation

[8] `https://keras.io/` Accessed: 2019-11-16

[9] The name sequential here only refers to the sequentially created and stacked layers.

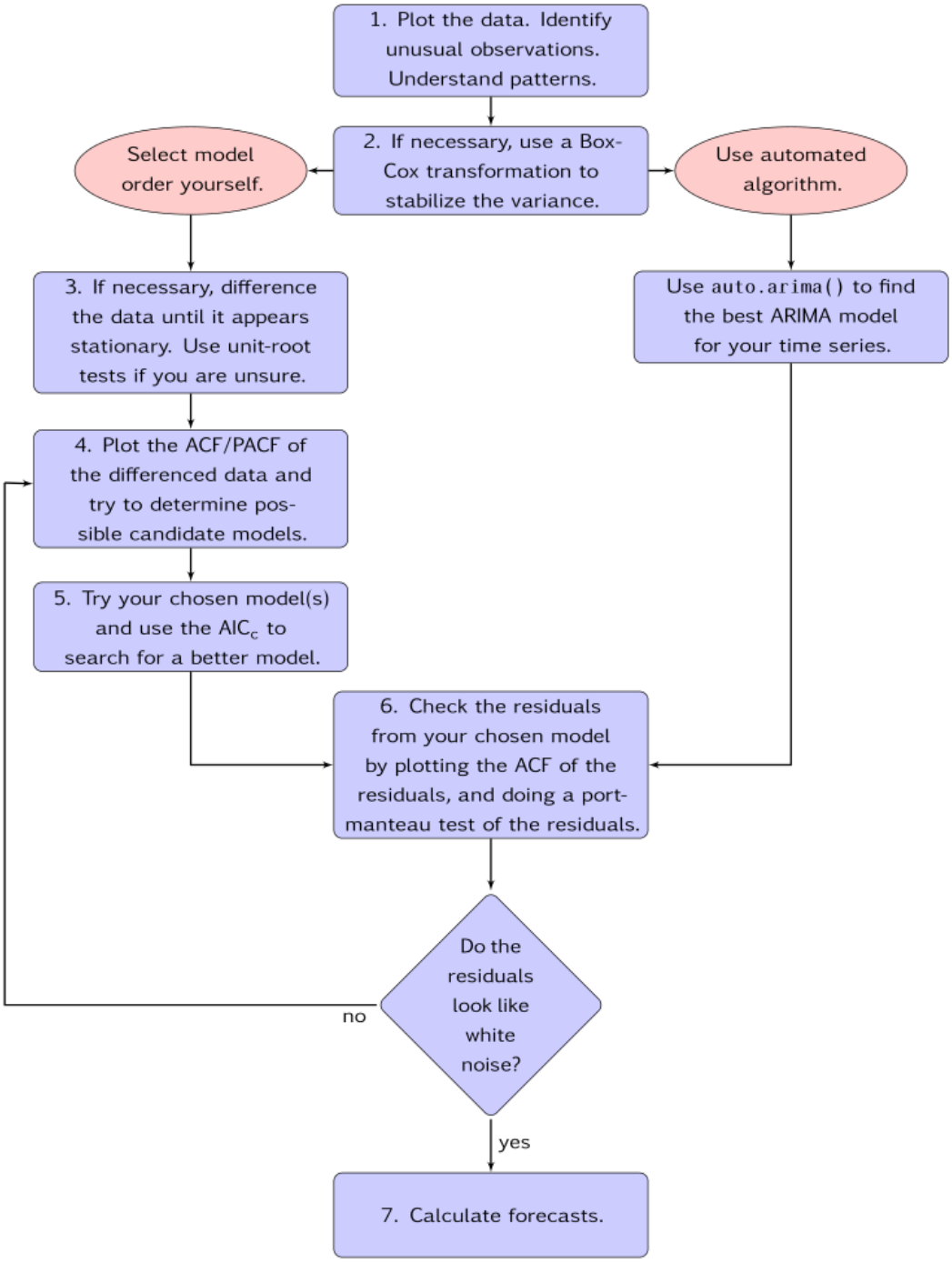


Figure 10: Basic procedure for identifying the order of the ARIMA model.[25]

Algorithm 1 Creating a sequiental Keras model

```
model = Sequential()
.
.
.
model.add(Some kind of Keras layer)
model.add(Some kind of Keras layer)
.
.
.
model.compile(loss=Loss Function, optimizer=Optimizer)
```

functions and so forth) as are needed for the model. The last statement must be the compilation of the model with a loss function and an optimizer. With this layer architecture, building a model in Keras is easy. However, what Keras is not helping one with is building a well functioning model. The different kind of layers, parameters at each layer, the loss function and the optimizer with all its parameters determine the prediction performance. Finding good values for these parameters and assessing the resulting prediction performance - in comparison with ARIMA - is the goal of this evaluation. What architecture and parameters were selected is described in Section 6.2.1

5.3.2 LSTM Implementation

The algorithms are based on an example from the blog *Machinelearningmastery.com*[10]. The first important function implemented for the LSTM model library is the function for testing for the optimal configuration. This function uses the Talos framework to test through all defined combinations. The basic functionality is defined in the algorithm 2.

At the creation of the library object in the code, one has to define the data that the object is using. In the first important step of this function, the data is extracted for the source, destination pair and the correct number of most

[10] `https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/` Accessed: 2019-11-21

Algorithm 2 Test_for_optimal_configuration *inputs* : *source_range*, *destination_range*, *number_of_values*, *parameter_variations*, *number_of_repetitions*

```
test_folder = create_test_result_folder()
for source in source_range do
  for destination in destination_range do
    data = get_sub_data(source, destination, number_of_values)
    data = scale_values(dataset, 0, 1)
    training_values, target_values = create_data_for_training(data)
    for repeat in number_of_repetitions do
      start_talos_scan(training_values, target_values, cre-
      ate_model_function, parameter_variations)
    end for
    move_test_data(test_folder)
  end for
end for
```

current values. This data is then scaled correctly between 0 and 1 and from this set, two arrays are created: one with training values and one with the target values. These values are then put into the Talos scan method, that tries all the previously defined parameter combinations. The *create_model_function* parameter passed to the Talos method is the function that creates the actual model, following the scheme defined in the previous section of how to build a Keras model with added placeholders so that Talos can insert the defined parameters.

The actual method for training one single model is similar to the configuration testing method; that is why I will not add the pseudocode here. It works the same with some differences. It just gets a model name it is supposed to train and no parameters it should try, and then it starts the Keras training method.

5.3.3 PmdARIMA (Automatic ARIMA parameter optimization)

PmdARIMA is a Python library that provides a possibility to determine the best ARIMA model for predicting a time-series automatically. For this thesis, I used the traffic of multiple communication pairs. For this reason, PmdARIMA provides a fast way of generating an ARIMA model without having to analyze every time-series in detail. The library provides a fast way of generating an ARIMA model to compare the performance of the LSTM predictions.

The optimization does not need many prerequisites. The parameters previously described in Section 5.2 p, d, q and P, D, Q can be set to a specific start value and a maximum that the optimization is not crossing; this can be used to optimize the search for the optimal parameters. Nevertheless, the optimization can also be started without configuration these values or any other value except one. The only thing that I had to set is a parameter m. This parameter sets the number of periods one season has, how the m parameter is used was described in section 5.2.1. Examples for this parameter would be:

- Quarterly data: m = 4
- Monthly data: m = 12
- Hourly data: m = 24

Important is that the data can contain multiple seasons. For example, the GÉANT set used in this thesis has 15-minute steps; this leaves multiple possible seasons:

- half-hour season: m = 2
- one hour season: m = 4
- one day season: m = 96
- one week season: m = 672

I tested all four possibilities. The test[11] showed that m = 4 yielded the model with the best prediction.

[11] Automatically fitted ARIMA model on the communication pair 1 to 11. Using one prediction to test the quality of the model.

The input for the ARIMA model is always an ordered time-series, no matter if it is an automatically fitted ARIMA model or manually fitted one.

An ARIMA model can only ever predict from the last point it was fitted on. So, for example, if a time series has the values [1,2,3,4,5] and we want to predict the 6th value, the ARIMA model needs to be fitted until the 5. How long the fitting sequence is, is up to the person fitting the model. The only thing that cannot be changed is that it has to end with the 5 in this example. From there, it can make any number of predictions, e.g. the next five steps, but only the first prediction will have a high quality, any prediction after that loses accuracy fast. What an ARIMA model also does; is that it calculates a confidence interval. The standard in the PmdARIMA library is a 95% confidence interval.

6 Evaluation

The evaluation chapter takes the theoretical background from the previous chapters and puts them to practical use. Here the parameters from the Long Short-Term Memory models get tested and analyzed for the best values. From these parameters a model is build and tested against an Auto-Regressive Integrated Moving Average model.

6.1 Test Data

As described in Section 4.2 the chosen dataset is the data from the GÉANT network. Figure 11 shows the topology of the network[12]. In the dataset all communication pairs are included. So communication between node 1 and 2, node 1 and 3, node 1 and 4 and so forth. As test data for the evaluation five of these communication pairs were used. The pairs are source 1 to 5 all to destination 11. I cannot show the exact nodes in Figure 11 as the data I have was anonymized by the authors. Also described in a previous Section (4.3) is how the data is interpreted. I interpreted nodes 1 to 10 as services while the rest of the nodes I interpreted as the ingress nodes. So, the five communication pairs are five different services accessing the node 11. Each pair has a distinctly different pattern of communication. The training data consists of 1199 values. The library uses 999 of those values for training and the rest used for testing the models. From the 999 values again, 20% is used for validating the training steps. This validation data is used to monitor training success. If both the loss on the training and validation data is improving (getting smaller), that means the training is working. If only the loss on the training data is improving, that means the model is overfitting on the training data[13]. To test the data after training, a third test set is used, that was not involved in the training of the model, to confirm the performance on the training and validation set. For the hyperparameter optimization, the number of tests per configuration was 40 repetitions. In the plots I removed the outliers, so the scale of the plots was

[12] `https://www.geant.org/Networks/Pan-European_network/Pages/GEANT_topology_map.aspx` Accessed: 2019-11-21

[13] This means that the weights of the model are fitted too much for the values of the training set and the model loses the ability to predict other data.

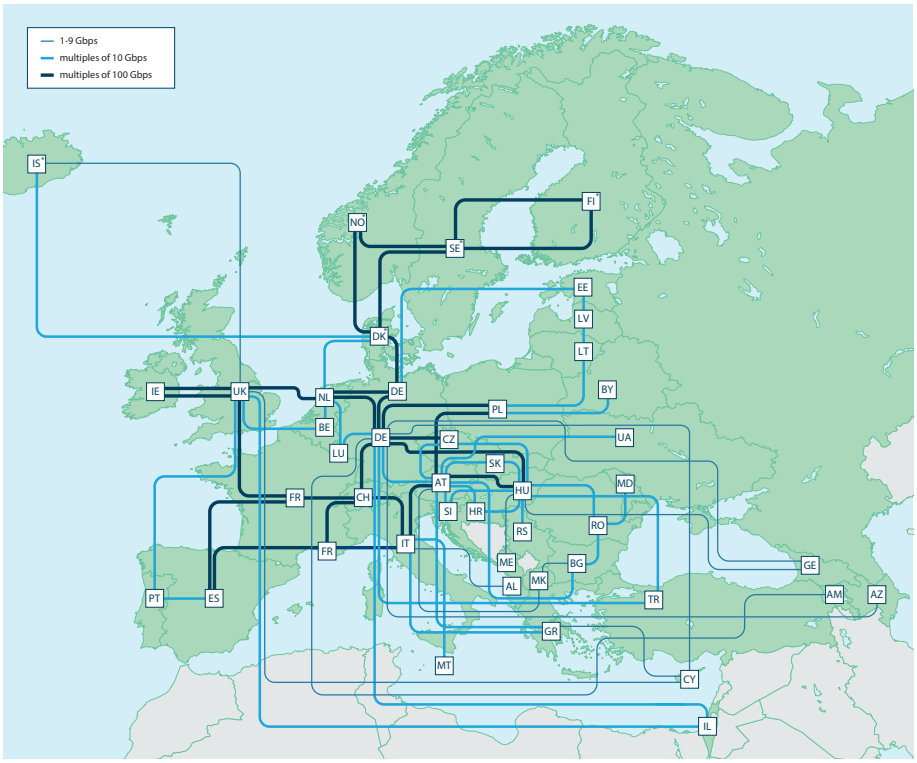


Figure 11: GÉANT network topology

not too big, but each box had not more than five outliers. The majority hat less than two outliers. All the experiments where the parameters combinations changed contain 40 repetitions.

6.2 The LSTM Model

The content of this section describes the construction of a model containing LSTM layers, how the data has to be modified, how to do hyperparameter optimization and what to look out for.

6.2.1 Identifying the best Architecture and Parameters

For constructing an LSTM model, Keras provides an LSTM layer. This layer implements an RNN LSTM layer with all the LSTM mechanisms all already implemented. Keras provides many configuration possibilities for each layer. The options for the LSTM layer include the number of hidden nodes, activation function, use of bias and so forth. Because of the many possible configurations and the resulting complexity, I decided to keep the standard values of most parameters and tune the parameters that mostly influence the complexity of the layer, which in the case of the LSTM layer is the number of hidden units.

Also combining multiple LSTM layers creates a network with higher computation complexity. First, the input and output are explained to understand better how to connect layers. The input for the first LSTM layer in the architecture has the shape of (batch size, time-steps, features). When retaking the example of the sequence [1,2,3,4,5] and using two values to predict the third value: our input-array when using the Keras would look like this:

$$[[1, 2]$$
$$[2, 3]$$
$$[3, 4]]$$

The pair 4,5 is left out because there exists no corresponding result. Keras splits the array apart and feeds the LSTM with the sequences of 1,2 / 2,3 / 3,4 , when each batch has the size of one. Of course, also fed into the model is the corresponding third number as a target for the training. Usually, the output of the LSTM would be of the shape (batch size, units) (units meaning

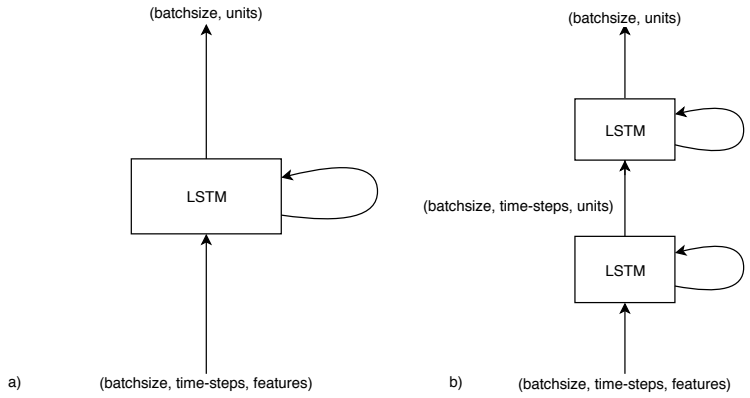


Figure 12: Examples of how Keras sets up LSTMs and how the inputs and outputs are shaped.

the hidden nodes), as can be seen in Figure 12 a). However, when constructing the network, one has to pay attention to setting the parameter *return_sequence* to true. This results in a new output with an extra dimension containing all predictions, not just the final one so that the LSTMs can connect as in Figure 8.

As described before, the output of an LSTM layer has the dimension (batch size, units). This is not the one value we want from the prediction. To get one definitive result in Keras the vector with the size of units has to be reduced to one output, which is done using a *Dense* layer. The *Dense* layer is a basic neural network layer that connects all inputs to a defined number of outputs, for example, one value.

As two examples for two implementations: the original paper that introduced LSTM did not yet have a forget gate [28], the forget gate was introduced later in [29].

As described in Section 5.1.3, the optimizer and loss function are also an essential factor in the training of a neural network. For the testing, I decided to stay with the *Mean Squared Error* loss function, as it is a simple and effective way of judging the quality of the training. Furthermore, I chose the ADAM optimizer, as it is also an often used and standard optimizer.

For the actual testing of the models, an extra library is used, called *Talos*. Apart from the hyper-parameter optimization, *Talos* has the bonus that the library stores all training results from each parameter combination in a .csv

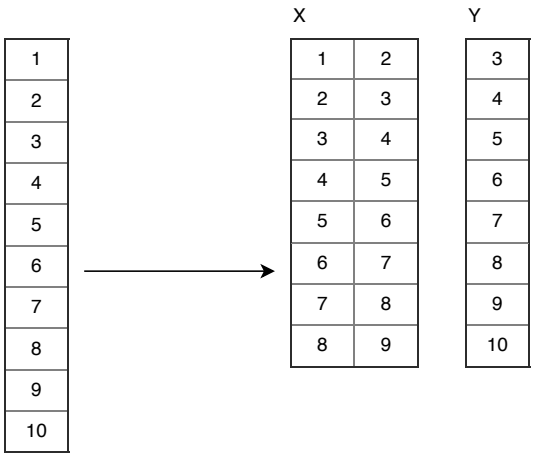


Figure 13: How the arrays are created for the training of an LSTM.

file for easy analysis.

6.2.2 Training the LSTM

The input of a neural network during the training is always an input array and a corresponding array with targets that the model should predict.

In the example in Figure 13 the sequence 1 to 10 is transformed into an input array X and a goal array Y. For this example the number of previous values used for a prediction is two. That is why the array X has two columns. Usually, the input array would need three dimensions for an RNN model in Keras. The third dimension would be the feature dimension. Since there exists only one feature, I omitted the dimension in this example.

6.2.3 Hyperparameter-Tuning Results

The next section describes the results from tests with hyperparameter optimization using the data described before.

Hidden Nodes First, the effect of the number of hidden nodes was analyzed. This parameter describes the number of hidden nodes per LSTM layer. As can be seen in Figure 14 increasing the number of hidden nodes results in a higher probability that the network will produce low error predictions. The same can also be seen with the validation loss. However, when increasing the epochs

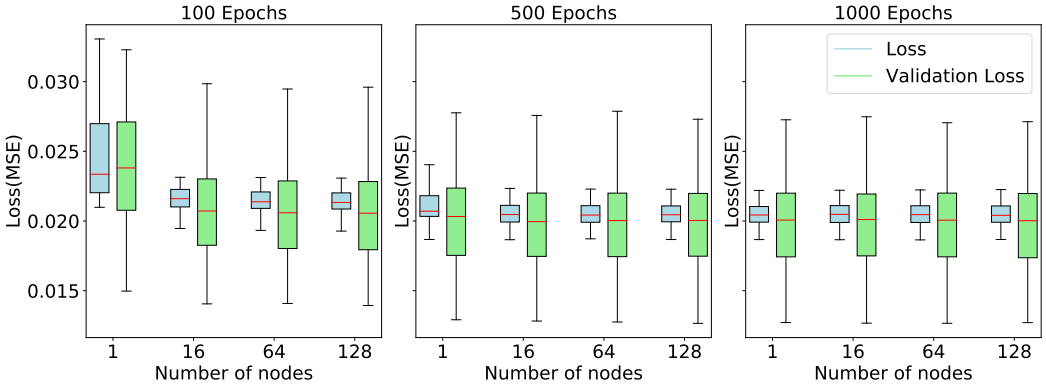


Figure 14: The loss and validation loss when increasing the number of hidden nodes for pair (1,11).

(more training steps), the advantage for the higher number of nodes vanishes and all models perform the same (Figure 14). The shrinking advantage is an important fact because the training time can increase drastically when using a high number of hidden nodes, especially when combining multiple layers. This pattern can also be seen with all the five different pairs Figure 28 to 30 in the Appendix. Except for the validation loss in Figure 27, here, the validation loss stays constant over the increasing number of hidden nodes and epochs. I suspect that the results look like this, since the model is only predicting one value. When using an output vector with more than one value and the resulting higher number of weights and more complex calculation hold only a small benefit when predicting only one value. Due to the unique traffic pattern of the pair (2,11), the pair is not reacting as well to the changes as the other time-series. I further analyze the pattern in section 6.5.

Previous Values Another important parameter is how many values to take into account when predicting a new value; utilizing one of the essential features of the LSTMs, the passing of prediction and cell-state to the next step as in Figure 8. When looking at the test in Figure 15, the advantage of using multiple previous values becomes apparent. For the validation set an advantage can be seen until using 4 previous values, after that the loss function stalls. Nevertheless, when increasing the epochs in Figure 15 it also becomes clear, that models with more than one step suffer from severe overfitting. The loss

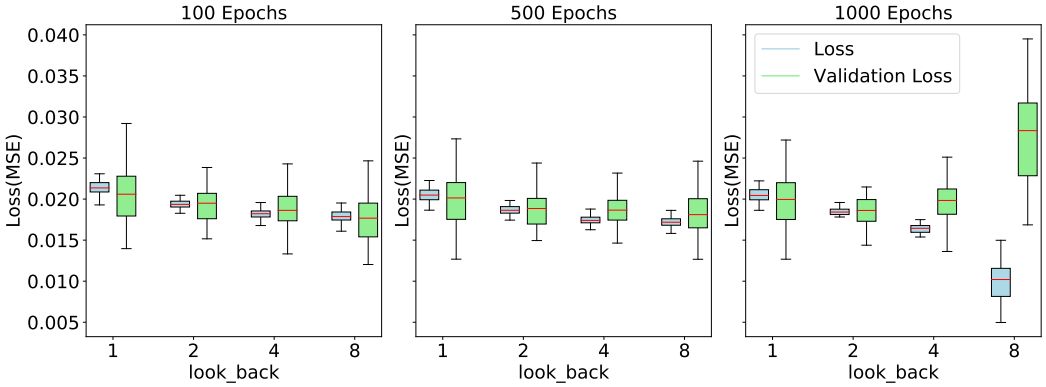


Figure 15: The loss and validation loss when increasing the number of previous values to take into account for a prediction for pair (1,11).

from the training set, trained with 1000 epochs, shrinks drastically, while the loss for the validation set increases. That shows that a model that uses multiple past steps is more sensitive to the training data. This could be because there exist fewer commonalities between the training data and the validation data and how they relate to their previous values. In short the rules that produce x_3 from x_1 and x_2 have only few correlations with the rules that produce y_3 from y_1 and y_2 with $y_3, y_2, y_1 \in ValidationSet$ and $x_3, x_2, x_1 \in TrainingSet$. So, these results show that when training the LSTM with multiple previous values, one must pay attention as it can lead to severe overfitting.

Number of Batches In Keras the parameter that controls the number of batches is called *batch_size*, this parameter indirectly controls the size of the batches by defining how many batches are created. So, through increasing the number of batches (fewer values per batch) in Figure 16, it is clear that with smaller batches you need a much higher number of epochs to get the same result. This occurs because in each epoch only one randomly chosen batch is shown to the model for training, so to get an equal coverage of the data it takes more epochs. The plots show that at some point smaller batches perform at least as good as the one big batch, but a lot more epochs are needed. However, one big batch also takes a lot longer to compute than one small batch per epoch, so a trade-off has to be found between computation time per epoch and number of epochs.

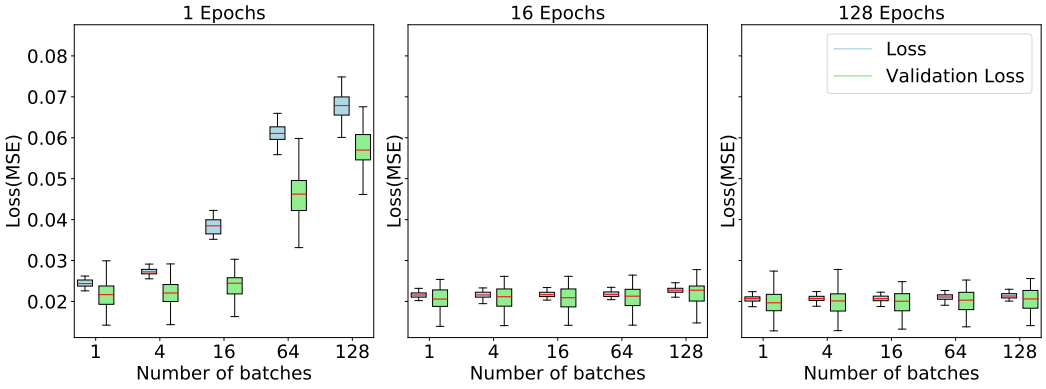


Figure 16: The impact of epochs vs. number of batches for pair (1,11).

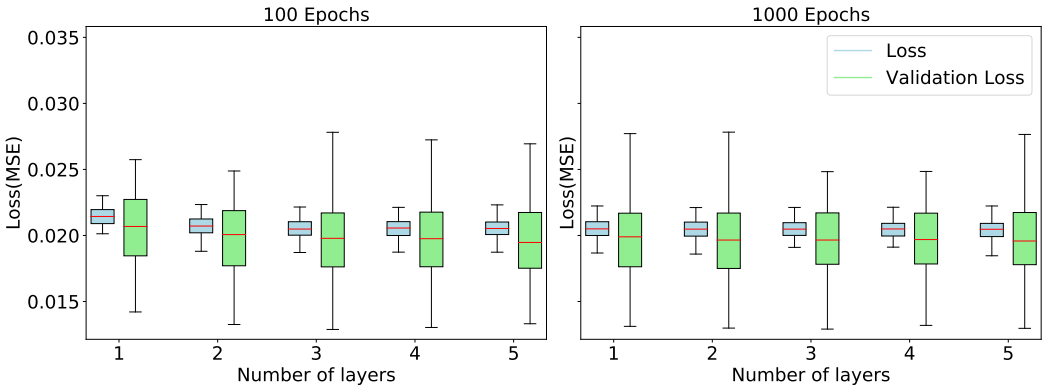


Figure 17: Plot of how the number of LSTM layers influence the loss and validation loss for pair (1,11).

LSTM Layers The last parameter tested was how the number of LSTM layers influence the prediction error; the layers are stacked just like in Figure 8. The plot in Figure 17 shows that more layers only slightly improve the predictions and and this is not the case for the other pairs (Figure 39 to Figure 42). This advantage also disappears further, when increasing the number of training epochs beyond 100. These plots show that in this application, the number of layers is playing a small role and one can completely disregard it when using enough epochs. This could be because the extra layers have the same number of hidden nodes in them like the previous ones and perform predictions based on predictions with the same complexity in each layer. A possible improvement could be the increase in complexity in the extra layers.

Now using all the test results, I can build a model that can predict the traffic. Since the test results were so similar, even one hyperparameter setting could work for all communication pairs. The plots in Figure 17 show that the model can use only one layer without it harming the predictions. The exception for this is the first communication pair. To get an equally good prediction for the first pair the number of epochs is chosen to be 1000. Also, the high number of epochs has only a negative effect when the model uses more than two previous values (look_back in the plot) , so the look_back is going to be two for the model. To speed up the training, since the model is training with 1000 epochs, I chose a batch size of 128. Furthermore, only one hidden node is selected, as more nodes do not bring an advantage.

Using this model trained for 1000 epochs produces results that Figure 18 shows. The rest can be seen in the Appendix in Figure 43 to 45. The test set contains the 200 most current values, and the models have never seen these values. The results vary from very good prediction (Figure 18b) to bad predictions (Figure 18a). What the results seem to have in common is that they are good at predicting the rough shape of the traffic. What all seem to struggle with is to get the exact values correct.

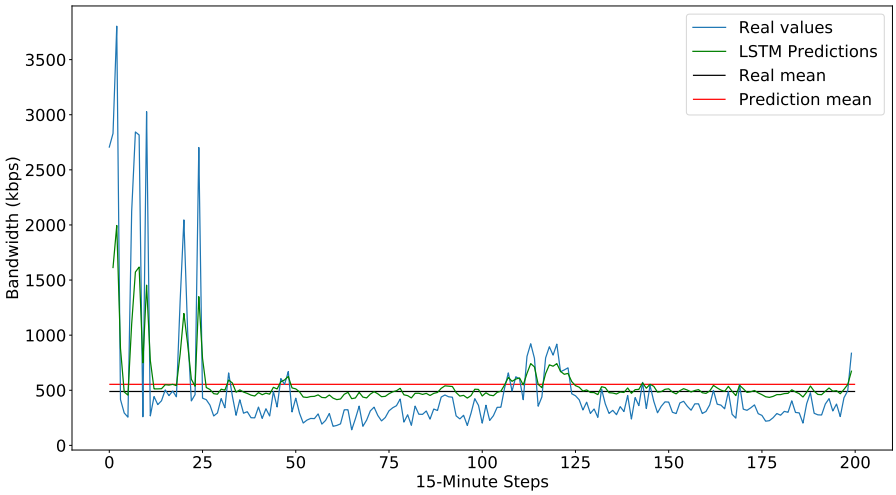
6.3 The ARIMA Model

Since I used the automatic ARIMA model builder, in this section, a short description is given on how the different chosen models look.

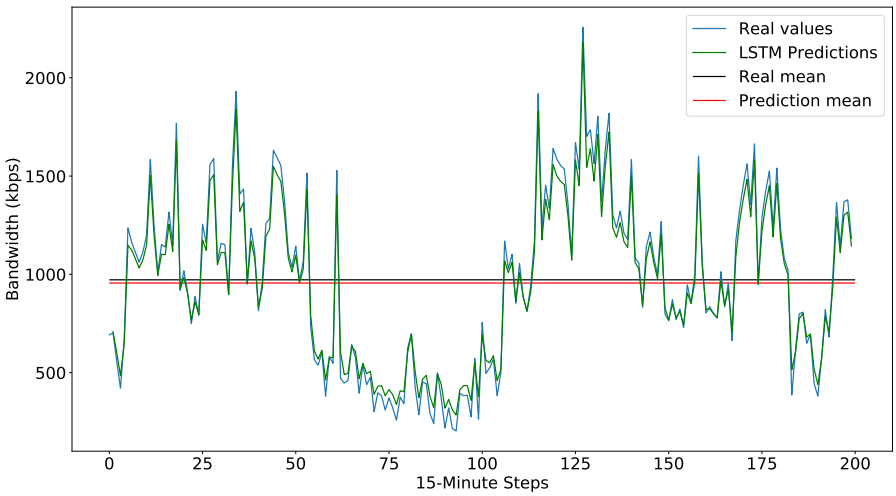
The values in the table below show the orders of the parts of the ARIMA model, chosen by the automatic ARIMA optimization.

source	p	d	q	P	D	Q
1	1	1	2	0	0	0
2	3	0	3	2	0	1
3	2	0	2	0	0	0
4	1	1	2	0	0	1
5	3	0	3	2	0	2

When using these automatically optimized models two of the predictions of those models can be seen in Figure 19, the rest can be seen in the Appendix



(a) Pair (2, 11)



(b) Pair (4, 11)

Figure 18: Prediction results of the traffic of pair (2,11) and (4,11).

in Figure 46 to 48. On first glance, these predictions look very similar to the LSTM predictions. Of course, there exist some obvious errors that the LSTMs are not producing like the overcompensation in Figure 19b at the 175th step. Nevertheless, the models predict other parts of the traffic equally well.

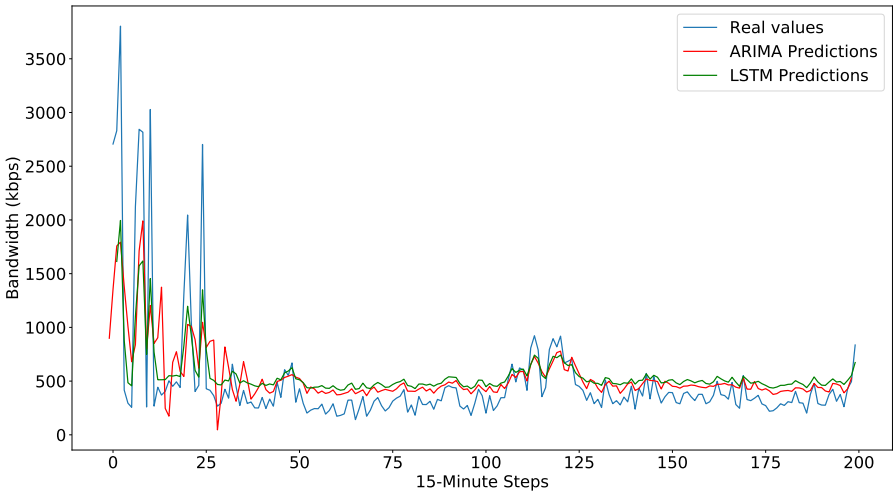
6.4 Comparing LSTM and ARIMA

Comparing all the errors from the ARIMA and the LSTM, the similarity in performance is undeniable. Figure 20 shows the average error in the predictions of the test set. Overall the LSTM is not much better than the ARIMA model. What could be said, is that the LSTM is more consistent as it produces fewer outliers than the ARIMA.

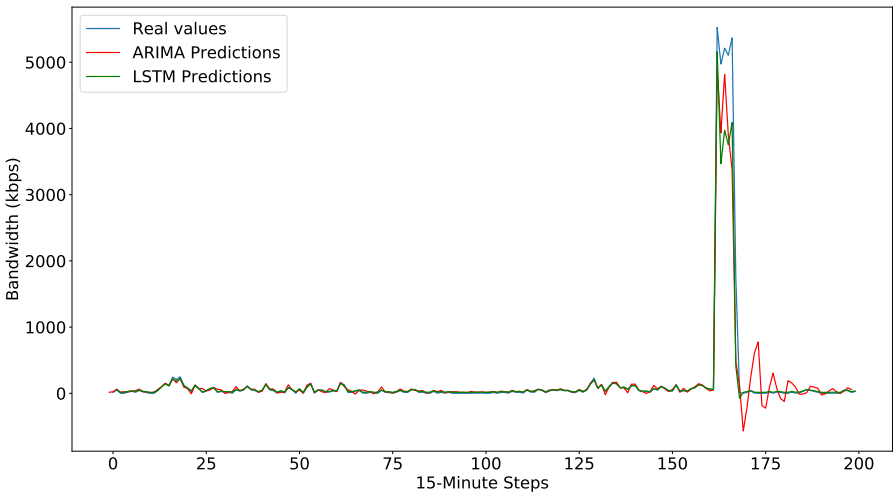
Also crucial for practical usage of the prediction models is how long they take to train and to predict a value. This dictates how often the system can make a prediction and how fast an algorithm could react to future changes in the traffic. Figure 21a shows the times for training an LSTM model and fitting an ARIMA model. The fitting of the ARIMA model is a lot faster than the training of the LSTM model. I trained the LSTM model with 1000 epochs, so using only 100 epochs would get the LSTM training time closer to the ARIMA time, but that could only be done if it would not impair the prediction quality.

The second plot, Figure 21b, shows the differences between how long it takes a model to predict one value. Included in the ARIMA time is the fitting of the model with the next value, as it is not usable if the model is not updated. Here the LSTM has a clear advantage to the ARIMA model. This is the case because for the LSTM the prediction is just calculating a few formulas with the previously found weights. For the ARIMA, on the other hand, it is not just the calculation of the formulas but the refitting of the model with the new data.

These two plots show that one has to make a careful choice. If the LSTM model needs to be retrained often, because the pattern of the data changes a lot, than the ARIMA has the clear advantage. Especially if the LSTM model would get more complicated, the time for training would increase. However, if the model does not have to be updated often, the LSTM performs much better than the ARIMA.

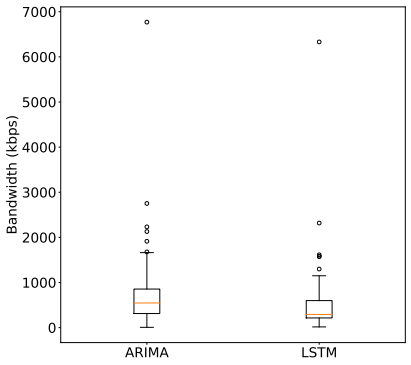


(a) Pair (2, 11)

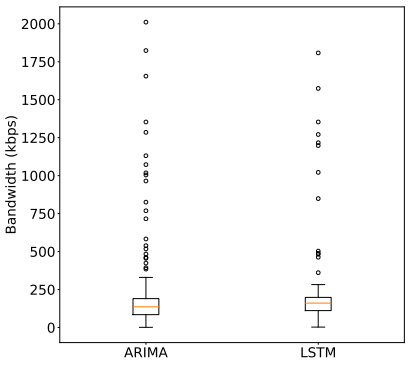


(b) Pair (5, 11)

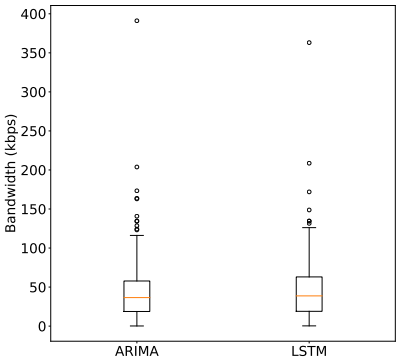
Figure 19: Predictions of the ARIMA model vs the LSTM model.



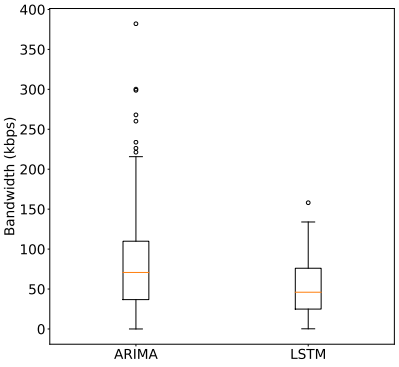
(a) Pair (1, 11)



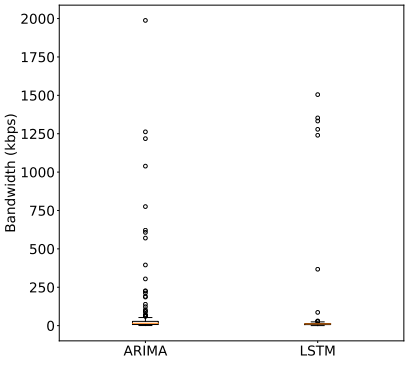
(b) Pair (2, 11)



(c) Pair (3, 11)

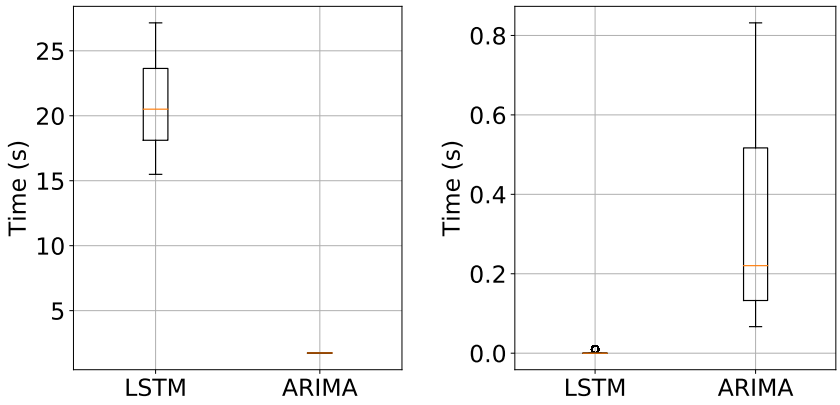


(d) Pair (4, 11)



(e) Pair (5, 11)

Figure 20: Comparison of the error in the prediction. ARIMA vs LSTM.



(a) The time needed for training/fitting the model

(b) The time needed for making a prediction of one value

Figure 21: Comparison of the relevant times measured for a LSTM and an ARIMA model

With just these results, I cannot recommend using the LSTM over the ARIMA. The performance on the prediction side is very similar, only decided by the lower number of outliers produced by the LSTM. When looking at the time comparison, the LSTM is slower to train but faster at predicting values. The only factor that distinguishes the two approaches is complexity. One can automatically compute the ARIMA model after finding a suitable parameter m, number of periods in a season. On the other hand, the LSTM model architecture allows for a vast number of different configurations and different parameter settings. Finding the best parameters can also be done automatically via a hyperparameter optimization, but depending on the number of parameters and the computation resources available, the optimization can take up to a few days to complete.

6.5 Optimizing LSTM Prediction

To analyze the predictions, I looked closer at the pair with the worst predictions, pair (2,11). When looking at the prediction in Figure 18a, the predictions never go below a specific value. Furthermore, the model predicts the majority of the values too high. Figure 22 shows the traffic of the pair (2,11) and

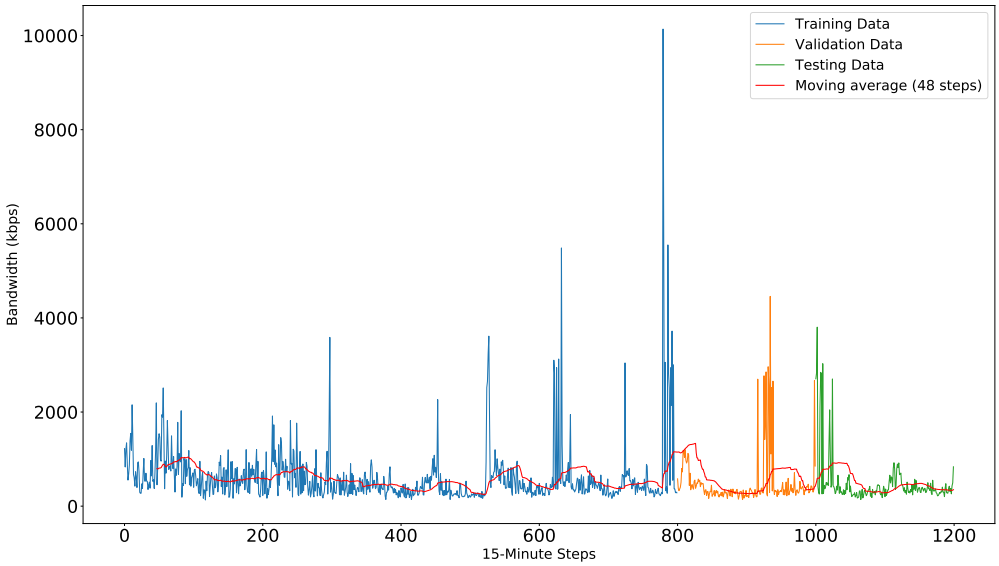


Figure 22: The traffic from pair (2,11) used for training, validating and testing the model.

how it was split up into training, validation and testing data. Also included in the Figure is the moving average over 48 steps, that is equivalent to 12 hours, so it is a 12-hour moving average. In the first part of the data from step 0 to 400 the moving average shows a decline with two small spikes but no discernible pattern. Looking further to the right after step 400, a pattern emerges. There are bumps in the moving average of the length of roughly 100 steps. The bumps continue until step 800 with a broader bump in the average. After that broader bump, it continues with the spikes of width 100. When seeing this pattern, maybe the prediction could be improved when omitting the first 400 steps that show no sign of the pattern. Additionally, to omitting the 400 steps, the test includes multiple data ranges in 100 step steps, training with 100 steps, 200 steps and continuing this to 600 steps. However, testing with the 600 steps that omit the 400 first steps has absolutely no influence on the prediction. The model performs the same as with the 1000 steps. What improved the model was training it with just 200 steps. As can be seen in Figure 23, the predictions get closer to the actual real values. This proves that the amount of training data fundamentally changes the ability of a model to make accurate predictions. Also, this makes the models more challenging to

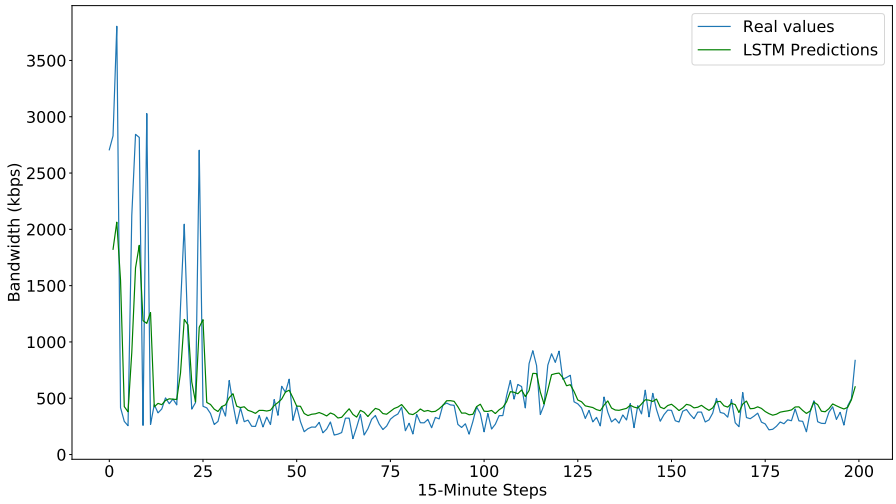


Figure 23: Prediction when LSTM is trained with 200 values.

use in a practical scenario as every model needs a different amount of data and a way of identifying how much data it needs. What is also very important is that training the model with only 200 values made the training a lot more inconsistent. The result in Figure 23 is one of the better results, but the training could also result in a worse performance than training with the 1000 values.

6.6 One-Step Prediction vs Multi-Step Prediction

In this section two approaches are closer described that the ARIMA model can not replicate. The first approach is the prediction of values more than one step in the future from the current value. As mentioned earlier the ARIMA model can predict multiple values into the future at once, but anything beyond one value ahead has no usable accuracy. Again using the example of the sequence [1,2,3,4,5], an example would be to predict the 4 based only on the sequence 1,2. The training of the LSTM works the same as with the one-step prediction. The input sequence stays the same, what changes are the target values. These values are shifted by the number of values one wants to predict in the future.

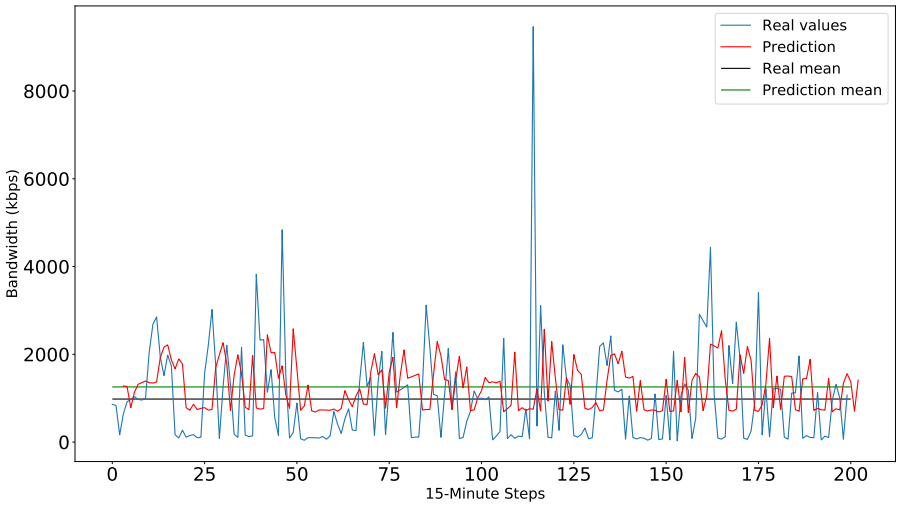
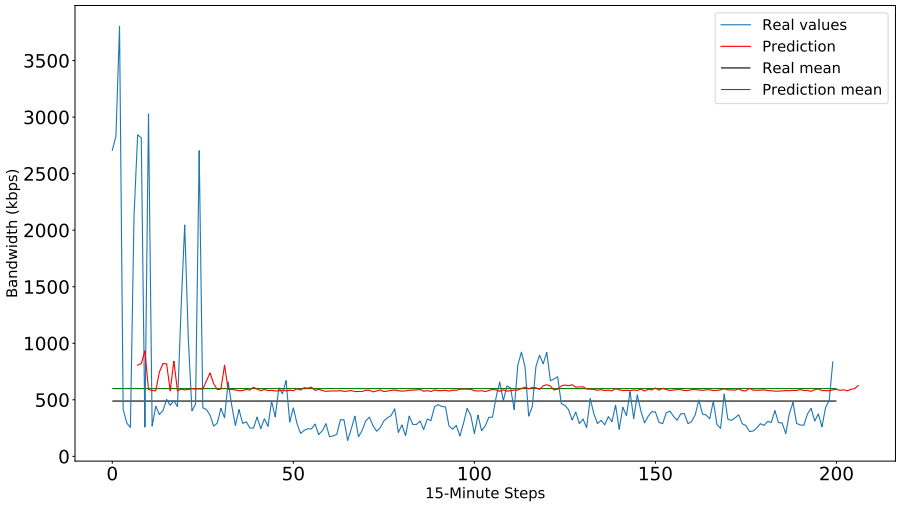


Figure 24: LSTM prediction, when trained with 4 steps into the future.

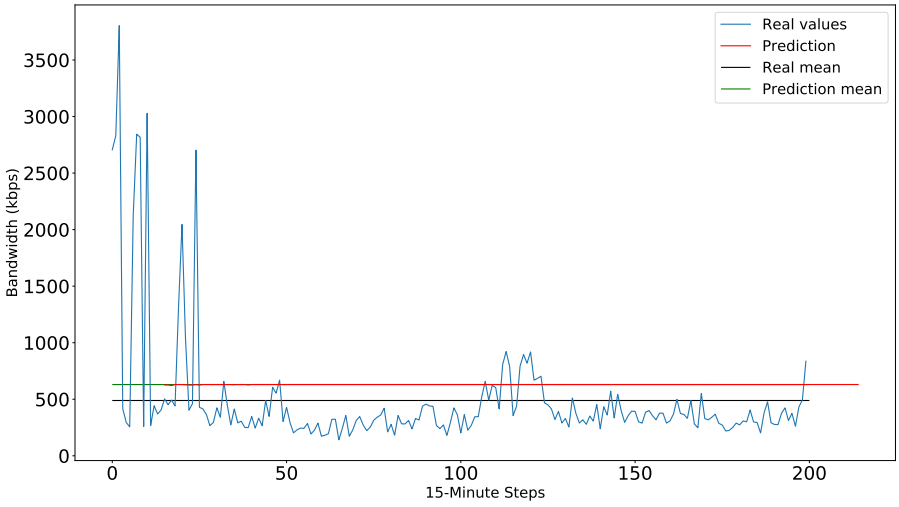
But what happens when the LSTM is trained this way is displayed in Figure 24. In this example the LSTM was trained to predict 4 values into the future with two previous values as input. So, using two values to predict the next value in 4 steps. But the LSTM not actually predicts the value in 4 steps but a shifted and squashed version of the next step. So when inputting the value n-1 and n, instead of predicting n+4 the model predicted n+1 in a scaled down version. This trend continues when predicting further into the future. The squashing of the prediction continues until it approaches the mean of the one-step prediction. In Figure 25 this trend can be observed as the prediction of 16 steps into the future represents the mean of the predictions one step into the future.

The same phenomenon can be observed when trying to teach the LSTM to predict a range of future values instead of just one value; for example predict the next five values. When taking the output of each prediction that should represent n+1, n+2, n+3 and so forth and plotting them all on the nth step the output looks like in Figure 26. The example predicted the 10 next values, only using the two previous values. But here the output also produces multiple versions of the same function, which are more squashed the further into the

future the prediction lies.



(a) 8 Steps into the future



(b) 16 Steps into the future

Figure 25: The prediction of an LSTM predicting 8 and 16 steps into the future.

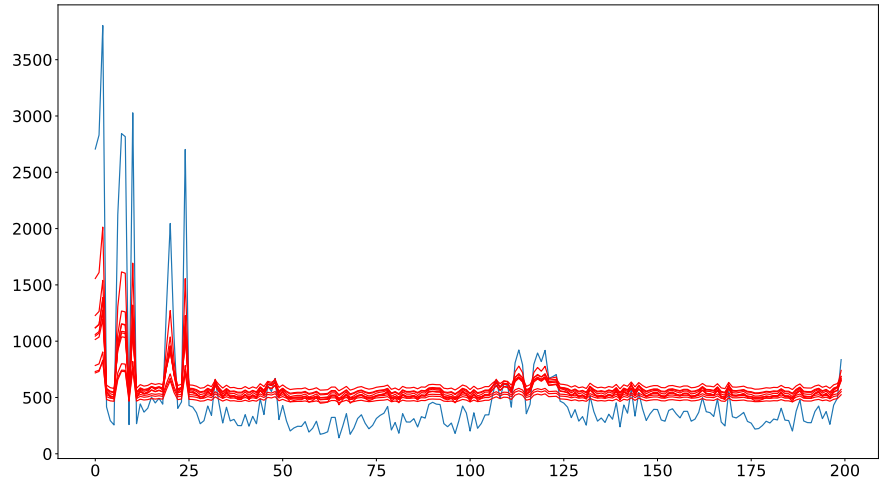


Figure 26: 10 values predicted at the same time plotted without shift.

7 Future Work and Conclusion

This last section describes possible approaches for building upon the results of this thesis. Furthermore, it analyzes and sums up the outcomes.

7.1 Future Work

One of the biggest possible additions to this thesis is testing the results on a live system. The live system would be perfect to find out how often the user has to retrain the Long Short-Term Memory model or how often the user has to initialize a new Auto-Regressive Integrated Moving Average model. When recording the live network traffic, the traffic can be put directly into the models and one can compare the predictions directly to the real traffic. Also, when putting this predicted traffic directly into a Virtual Network Function scaling and placing algorithm, the behaviour of the algorithm can be analyzed. This behaviour can then lead to first estimates what actual errors are acceptable and at what value the error is growing too big.

Since the performance of the LSTM was so similar to the ARIMA model, one should try to analyze other RNN layers. Keras supports GRU (Gated Recurrent Units) and ConvLSTM2D (Convolutional LSTM 2-Dimensional) layers. The works [6] [8], already mentioned in the Related Work chapter, both successfully use convolutional RNNs.

Additionally, on-the-fly-learning could be a promising strategy. When using each new step to directly continue the training of the model, the retraining of the model could possibly be eliminated. However, one would have to check if a training step after each new value is the optimal way or if the system has to wait until a whole new batch is complete. When using this approach, other layers likely have to be included in the model. I surmise that, because the system trains the model with much more data, a dropout layer would have to be included to prevent overfitting. A dropout layer drops some of the data that passes through it: how much data the layer is dropping can be configured. Furthermore, other layers also have a dropout parameter (the LSTM layer, for example). This would have to be included in the experiments as well.

Additionaly the size of the steps may play a role in the prediction quality. The Abilene set [17], [18], has just 5-Minute steps. Maybe the 15-Minute steps

have less correlation as they pack more data into one step. That should also be tested and in a live system the step size could be chosen arbitrarily. Also a point that calls for tests with a live system.

7.2 Conclusion

The goal of this thesis was to analyze if an Long Short-Term Memory model is a viable approach to predicting network traffic in a Network Function Virtualization scenario. And this goal was fulfilled in part. The prediction models work well. When putting traffic in, a usable prediction for the next step can be achieved. The predicted value can be interpreted in two parts: the change from the previously predicted value and the actual traffic value. The reliable part of the prediction is the shape of the prediction, does the traffic go up or down in relation to the previously predicted value. With this prediction the VNF scaling algorithm should be able to correctly scale the VNFs in the correct direction, up or down. The other part of the prediction the actual network traffic causes the problems. Depending on the traffic pattern the actual predicted level of traffic can be off of the actual level. So, when putting an LSTM model before the placing and scaling algorithm it will be able to work with traffic predicted one step into the future. With these predictions the scaling of the Virtual Network Functions will go in the correct direction, but depending on the traffic pattern the predicted traffic bandwidth will be wrong. However, all these statements are also true for an Auto-Regressive Integrated Moving Average model. With these results it should not matter, for the VNF scaling algorithms, if an LSTM or ARIMA model is used. Maybe with additional test in different scenarios, as described in the previous section, a clear destinction can be made.

With these results I conclude that traffic prediction is a viable solution for NFV scenarios. Not neccessarily a deep learning approach like an LSTM but ARIMA models can already greatly improve VNF scaling and placing algorithms. The reuslts show that these algorithms can be changed from reactive to proactive with machine learning models.

8 Appendix

8.1 Documents and Code

In addition to the thesis document, python code is also part of this master's thesis. The main library is the *TimeSeriesModelCreator_Parallel_talos.py* file. This can be used to create LSTM prediction models.

Examples of the usage of the library can be found in the experiment files. An example is experiment one.

```
#!/usr/bin/env python

from TimeSeriesModelCreator_Parallel_talos import
        TimeSeriesModelCreator_Parallel_talos
import pandas as pd
import matplotlib.pyplot as plt

look_backs = [1]
modelCreators = []
for look_back in look_backs:
        modelCreators.append(TimeSeriesModelCreator_Parallel_talos
        (look_back,
        r'..\Datasets\GEANTCombined\all_in_one_complete_appended.csv'))

for modelCreator in modelCreators:
        batch_sizes = [100]
        epochs = [100,250,500,750,1000]
        nodes = [1, 2, 3, 8, 16, 32, 64, 128]
        layers = [1]
        optimizers = ['adam']
        losses = ['mean_squared_error']
        modelCreator.test_for_optimal_config('Epxeriment_1_1', 1, 5, 11, 11,
                1000, batch_sizes, epochs, nodes, layers, optimizers, losses,
                40,     1, 0, 200)
```

This is an example for testing multiple parameter variations. From the results of this optimization a model can be created and the model is saved in the library object. Where it can be called using its name. An example for this can be found in the Jupyter notebooks comparing the LSTM and the ARIMA models. The following example creates five different models and uses them to predict traffic.

```
creator = TimeSeriesModelCreator_Parallel_talos(2,
        r'..\Datasets\GEANTCombined\all_in_one_complete_appended.csv')
modelMatch = {}
for x in range(1,6):
        creator.add_new_model(name = 'test'+str(x), nodes = 1, layer = 1,
```

```
            loss='mean_squared_error', optimizer='adam')
        modelMatch[str(x)+'_11'] = 'test'+str(x)
        creator.train_model(1, 5, 11, 11, 1000, 200, modelMatch,
            epoch = 1000, batch_size = 128, shift = 0)

LSTMpredictions = []
for x in range(1,6):
        prediction = creator.predict('test'+str(x), subsets_testing[x-1], 0)
        LSTMpredictions.append(prediction)
```

These two examples cover most of the functionallities of the library. More examples and the library itself can be found in the *source/Training* folder.

8.2 Libraries and Frameworks

- Python 3.6.1
- Keras 2.2.5
- TensorFlow 1.14.0
- Talos 0.6.3
- Pandas 0.24.2
- sklearn 0.21.3

8.3 Figures

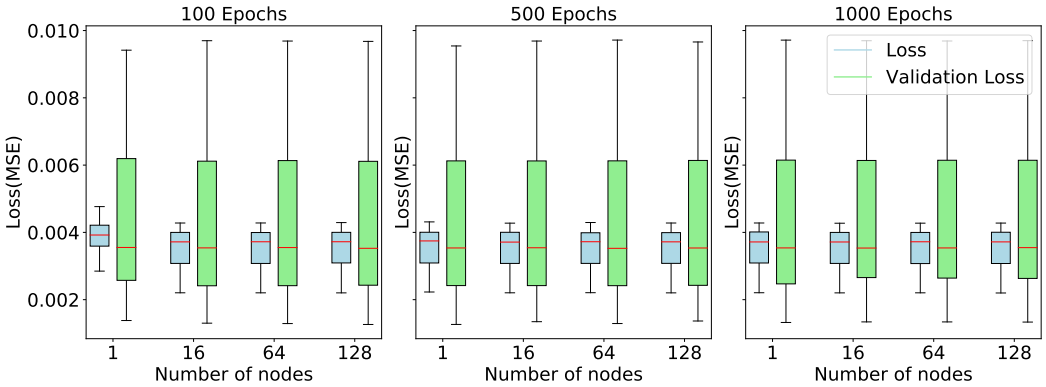


Figure 27: The loss and validation loss when increasing the number of hidden nodes. Pair (2, 11)

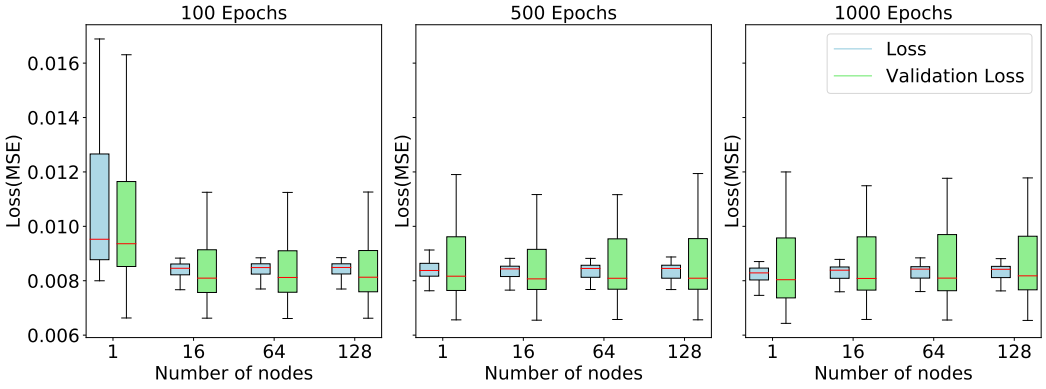


Figure 28: The loss and validation loss when increasing the number of hidden nodes. Pair (3, 11)

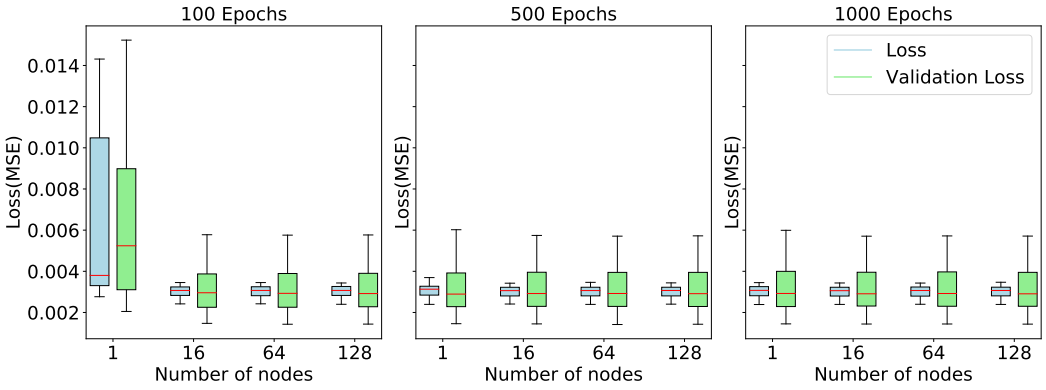


Figure 29: The loss and validation loss when increasing the number of hidden nodes. Pair (4, 11)

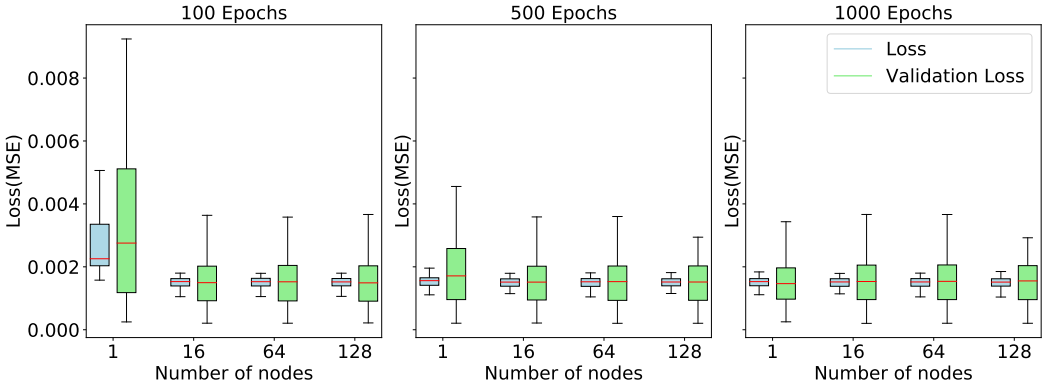


Figure 30: The loss and validation loss when increasing the number of hidden nodes. Pair (5, 11)

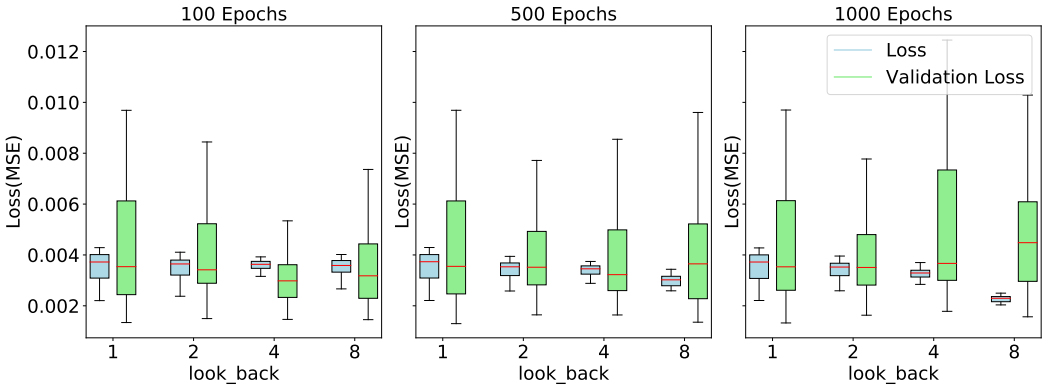


Figure 31: The loss and validation loss when increasing the number of previous values to take into account for a prediction. Pair (2, 11)

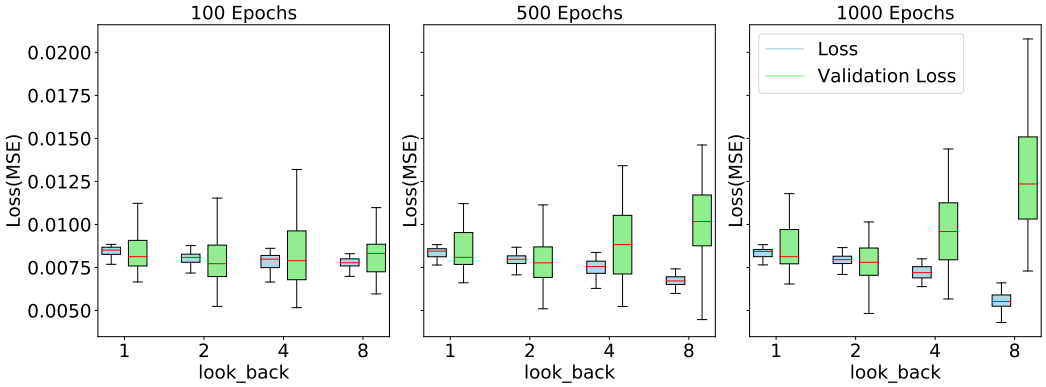


Figure 32: The loss and validation loss when increasing the number of previous values to take into account for a prediction. Pair (3, 11)

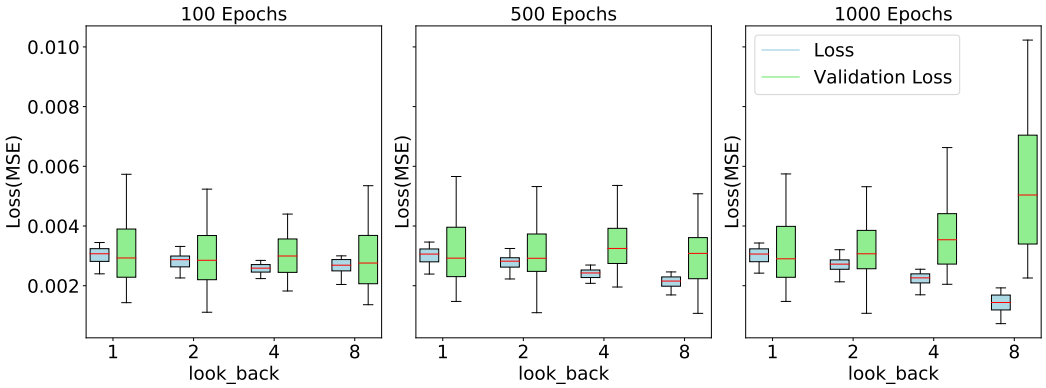


Figure 33: The loss and validation loss when increasing the number of previous values to take into account for a prediction. Pair (4, 11)

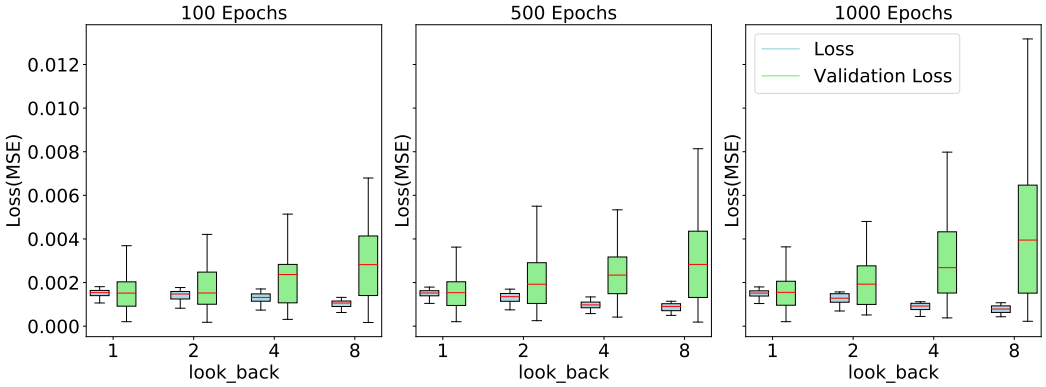


Figure 34: The loss and validation loss when increasing the number of previous values to take into account for a prediction. Pair (5, 11)

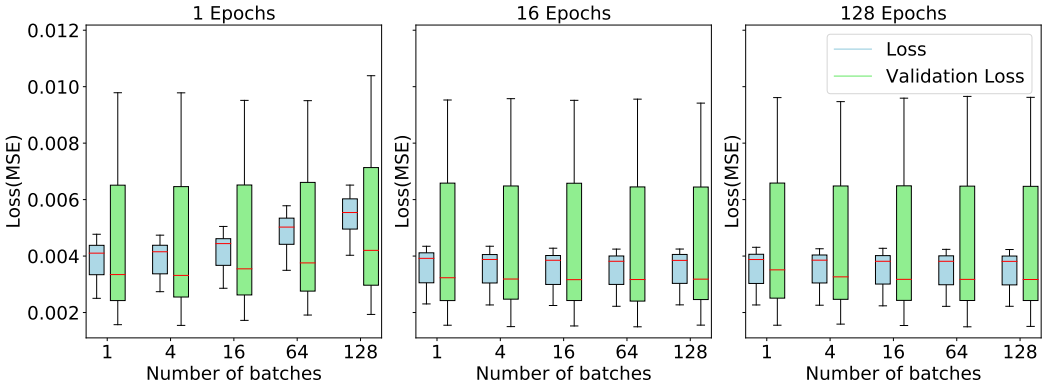


Figure 35: The impact of epochs vs. number of batches. Pair (2, 11)

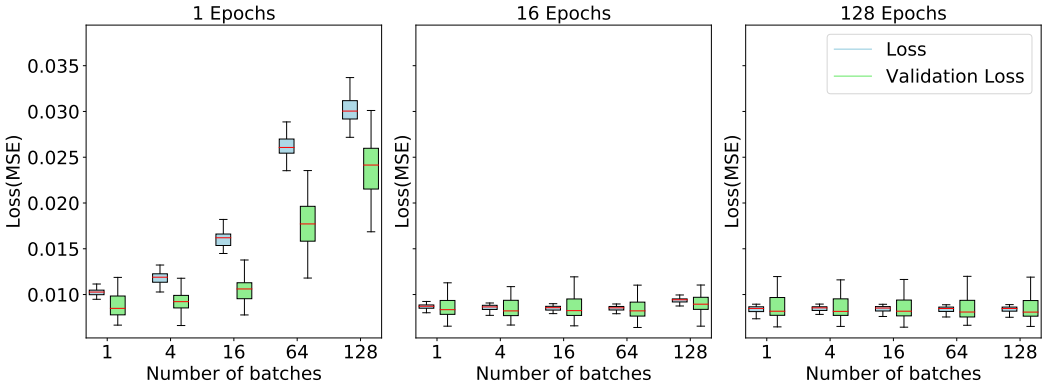


Figure 36: The impact of epochs vs. number of batches. Pair (3, 11)

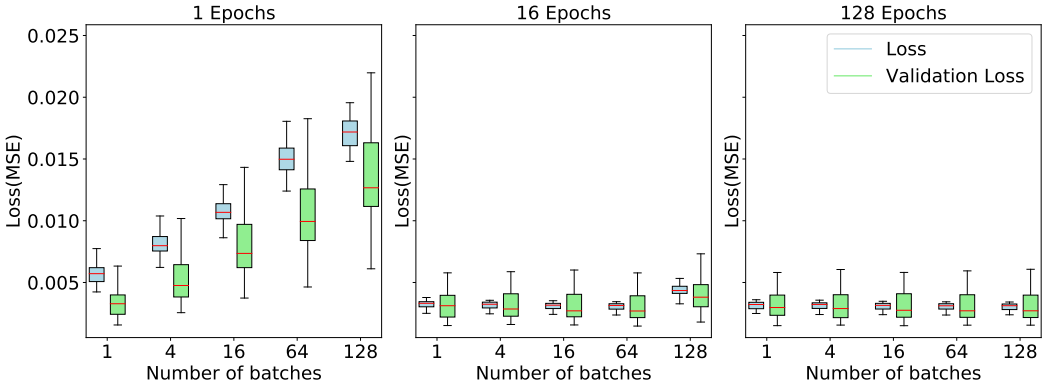


Figure 37: The impact of epochs vs. number of batches. Pair (4, 11)

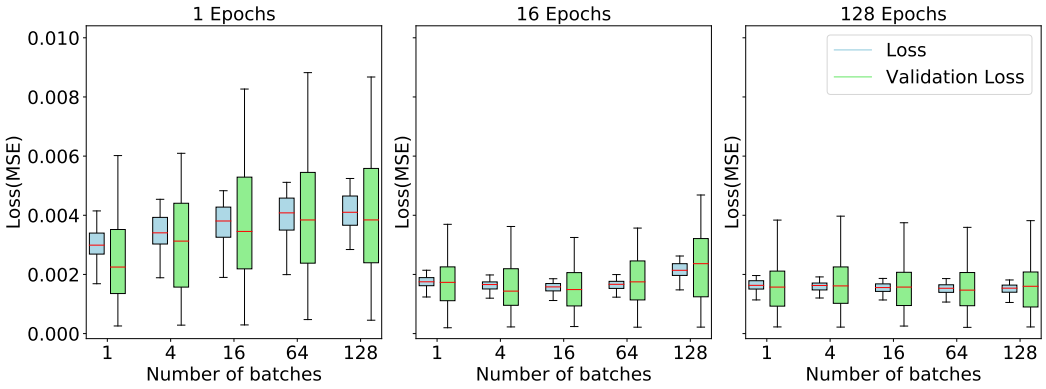


Figure 38: The impact of epochs vs. number of batches. Pair (5, 11)

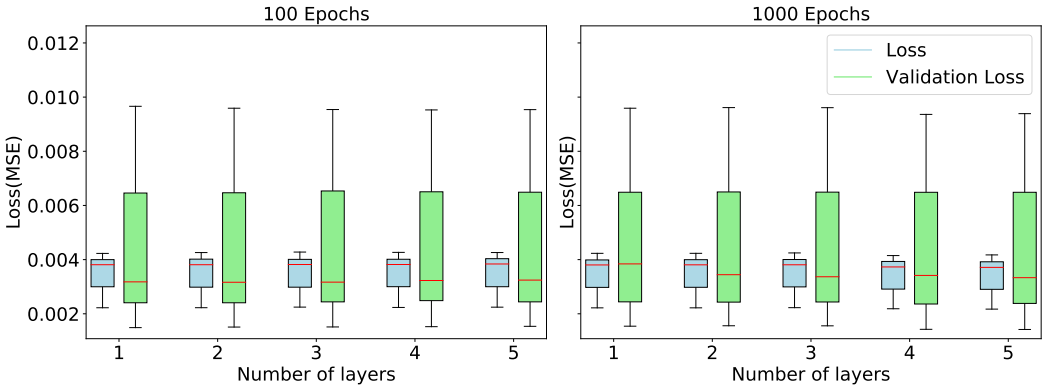


Figure 39: Plot of how the number of LSTM layers influence the loss and validation loss. Pair (2, 11)

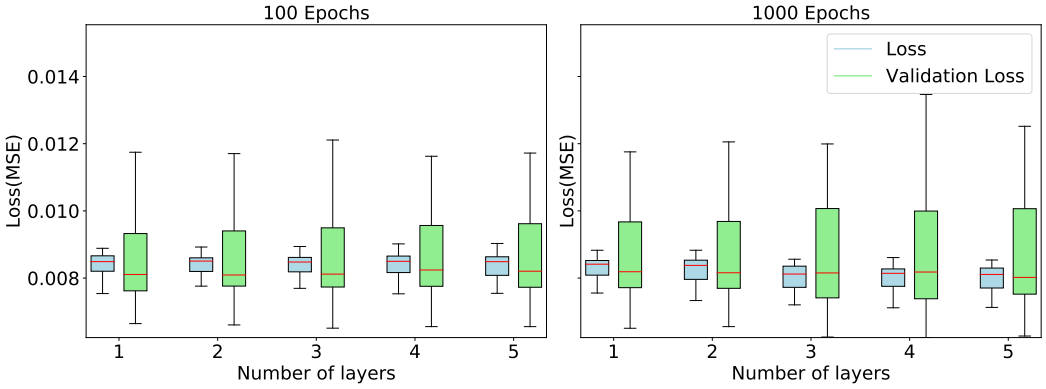


Figure 40: Plot of how the number of LSTM layers influence the loss and validation loss. Pair (3, 11)

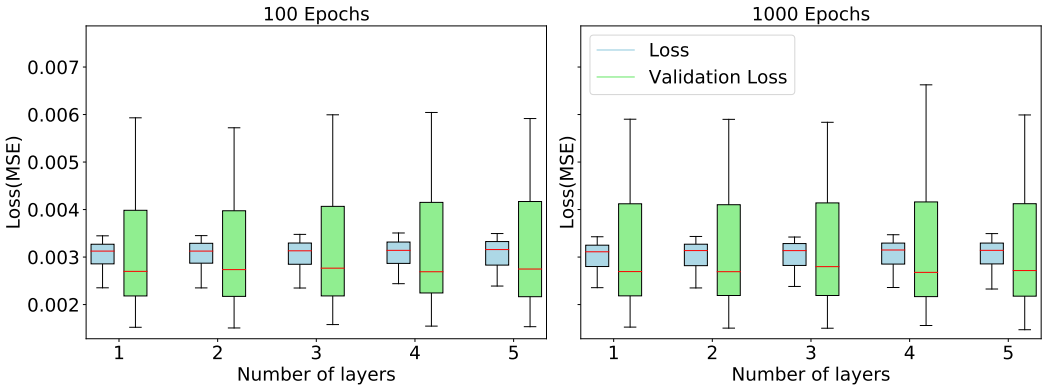


Figure 41: Plot of how the number of LSTM layers influence the loss and validation loss. Pair (4, 11)

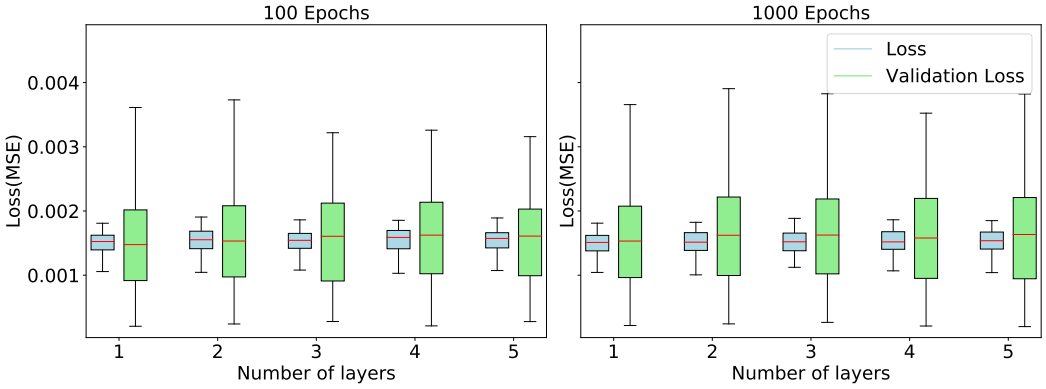


Figure 42: Plot of how the number of LSTM layers influence the loss and validation loss. Pair (5, 11)

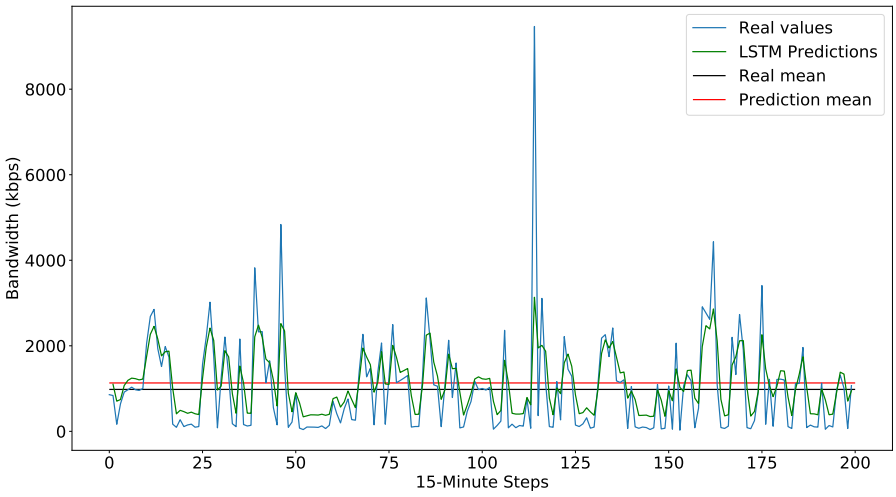


Figure 43: LSTM prediction of pair (1, 11)

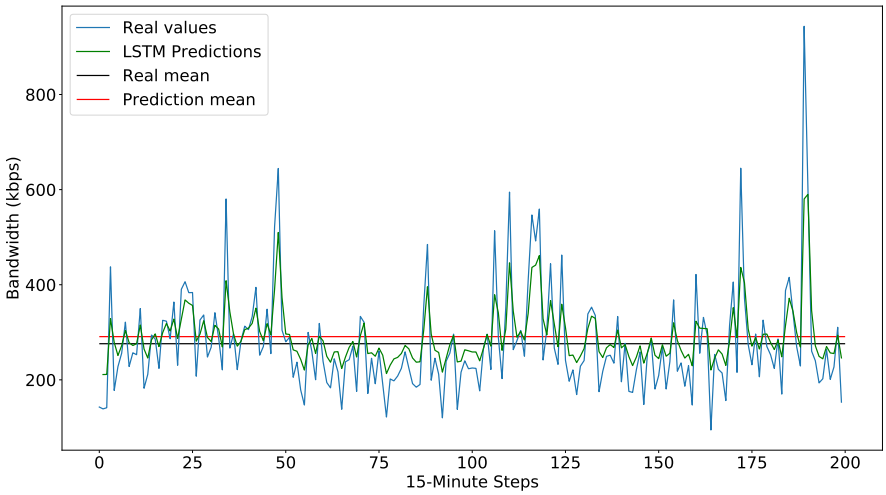


Figure 44: LSTM prediction of pair (3, 11)

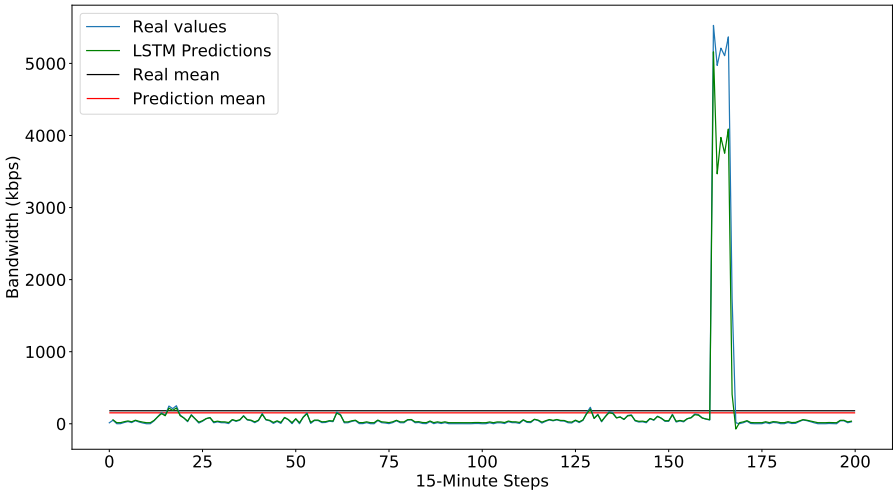


Figure 45: LSTM prediction of pair (5, 11)

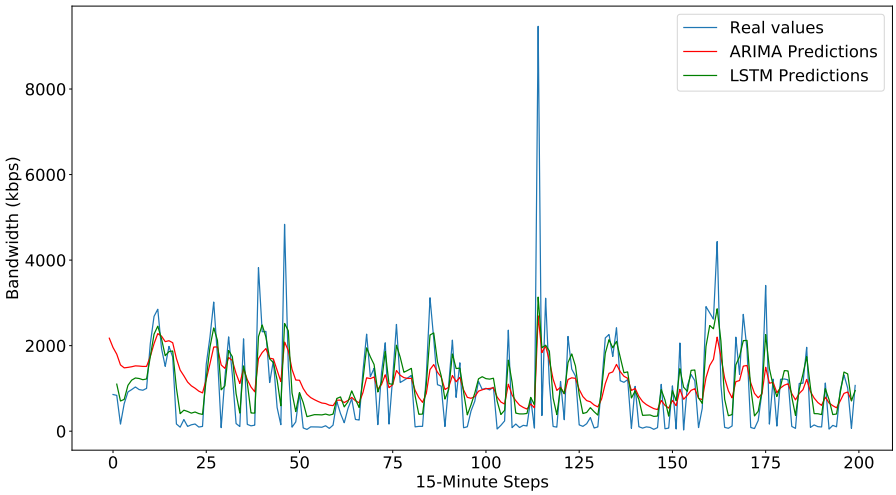


Figure 46: LSTM and ARIMA predictions of pair (1, 11)

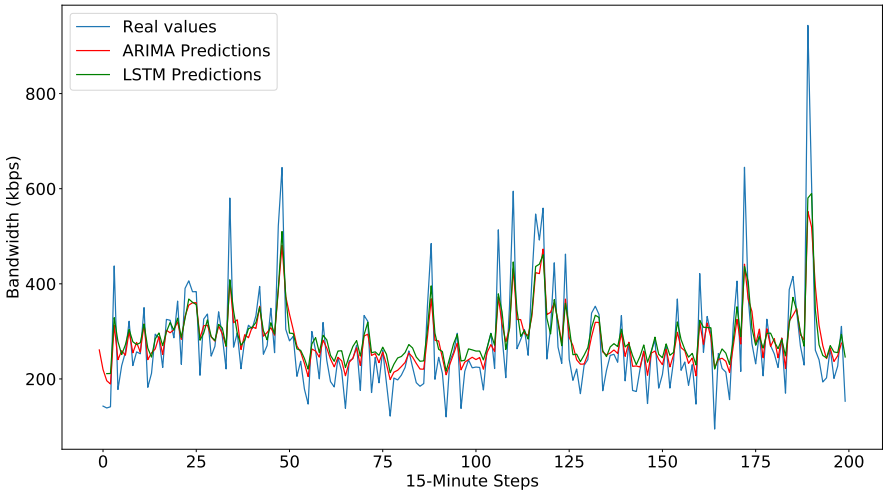


Figure 47: LSTM and ARIMA predictions of pair (3, 11)

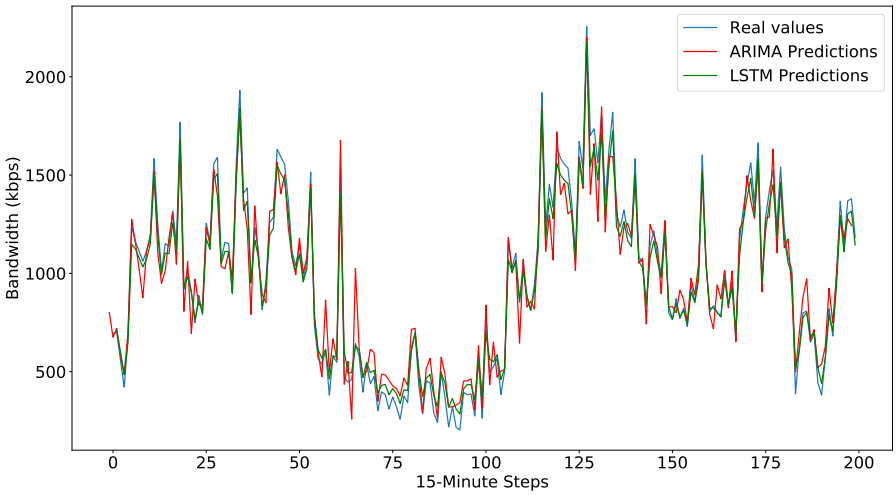


Figure 48: LSTM and ARIMA predictions of pair (4, 11)

References

[1] B. Yi, X. Wang, K. Li, S. k. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212 – 262, 2018.

[2] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba, "Topology-aware prediction of virtual network function resource requirements," *IEEE Transactions on Network and Service Management*, vol. 14, pp. 106–120, March 2017.

[3] S. Dräxler, S. Schneider, and H. Karl, "Scaling and placing bidirectional services with stateful virtual and physical network functions," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pp. 123–131, June 2018.

[4] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive vnf scaling and flow routing with proactive demand prediction," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pp. 486–494, April 2018.

[5] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, pp. 171–177, Oct 2015.

[6] V. A. Le, P. Le Nguyen, and Y. Ji, "Deep convolutional lstm network-based traffic matrix prediction with partial information," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 261–269, April 2019.

[7] R. Vinayakumar, K. P. Soman, and P. Poornachandran, "Applying deep learning approaches for network traffic prediction," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 2353–2358, Sep. 2017.

[8] C. Huang, C. Chiang, and Q. Li, "A study of deep learning networks on mobile traffic forecasting," in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pp. 1–6, Oct 2017.

[9] I. Alawe, A. Ksentini, Y. Hadjadj-Aoul, and P. Bertin, "Improving traffic forecasting for 5g core network scalability: A machine learning approach," *IEEE Network*, vol. 32, pp. 42–49, November 2018.

[10] J. Guo, Y. Peng, X. Peng, Q. Chen, J. Yu, and Y. Dai, "Traffic forecasting for mobile networks with multiplicative seasonal arima models," in *2009 9th International Conference on Electronic Measurement Instruments*, pp. 3–377–3–380, Aug 2009.

[11] H. Zare Moayedi and M. A. Masnadi-Shirazi, "Arima model for network traffic prediction and anomaly detection," in *2008 International Symposium on Information Technology*, vol. 4, pp. 1–6, Aug 2008.

[12] K. Kawashima, T. Otoshi, Y. Ohsita, and M. Murata, "Dynamic placement of virtual network functions based on model predictive control," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pp. 1037–1042, April 2016.

[13] H. Jmila, M. I. Khedher, and M. A. El Yacoubi, "Estimating vnf resource requirements using machine learning techniques," in *Neural Information Processing* (D. Liu, S. Xie, Y. Li, D. Zhao, and E.-S. M. El-Alfy, eds.), (Cham), pp. 883–892, Springer International Publishing, 2017.

[14] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos, "Knowledge-defined networking," *SIGCOMM Comput. Commun. Rev.*, vol. 47, pp. 2–10, Sept. 2017.

[15] W. B. de Vries, R. Van Rijswijk-Deij, P. de Boer, and A. Pras, "Passive observations of a large dns service: 2.5 years in the life of google," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*, pp. 1–8, June 2018.

[16] J. S. Rojas, Á. R. Gallón, and J. C. Corrales, "Personalized service degradation policies on ott applications based on the consumption behavior

of users," in *Computational Science and Its Applications – ICCSA 2018* (O. Gervasi, B. Murgante, S. Misra, E. Stankova, C. M. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, E. Tarantino, and Y. Ryu, eds.), (Cham), pp. 543–557, Springer International Publishing, 2018.

[17] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0–Survivable Network Design Library," in *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, April 2007. http://sndlib.zib.de, extended version accepted in Networks, 2009.

[18] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0–Survivable Network Design Library," *Networks*, vol. 55, no. 3, pp. 276–286, 2010.

[19] C. C. Aggarwal, *Neural Networks and Deep Learning*. Springer International Publishing, 2018.

[20] L. Medsker and L. C. Jain, eds., *Recurrent Neural Networks: Design and Applications*. CRC Press, 1999.

[21] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013.

[22] B. A. Pearlmutter, "Gradient calculations for dynamic recurrent neural networks: a survey," *IEEE Transactions on Neural Networks*, vol. 6, pp. 1212–1228, Sep. 1995.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[24] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, "Time series analysis : forecasting and control," 2008.

[25] R. Hyndmanm and G. Athanasopoulos, *Forecasting: principles and practice*. Melbourne, Australia: OTexts, 2nd edition ed., 2018.

[26] P. J. Brockwell and R. A. Davis, *Introduction to Time Series and Forecasting*. Springer-Verlag New York, 2 ed., 2002.

[27] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. Wiley, 5th ed., 2015.

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[29] F. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," *IET Conference Proceedings*, pp. 850–855(5), January 1999.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel

Machine Learning-Based Load Prediction for Network Function Virtualization Scenarios

selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum

Unterschrift