DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

Time Series Anomaly Detection and Uncertainty Estimation using LSTM Autoencoders

SARAH BERENJI ARDESTANI

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Time Series Anomaly Detection and Uncertainty Estimation using LSTM Autoencoders

SARAH BERENJI ARDESTANI

Master in Computer Science
Date: August 11, 2021
Supervisor: Hossein Azizpour
Examiner: Hedvig Kjellström
School of Electrical Engineering and Computer Science
Host company: Telia
Swedish title: Anomaliupptäckande i tidsserier och osäkerhetsestimering med hjälp av LSTM Autoencoders

Abstract

The goal of this thesis is to implement an anomaly detection tool using LSTM autoencoder and apply a novel method for uncertainty estimation using Bayesian Neural Networks (BNNs) based on a paper from Uber research group [1]. Having a reliable anomaly detection tool and accurate uncertainty estimation is critical in many fields. At Telia, such a tool can be used in many different data domains like device logs to detect abnormal behaviours.

Our method uses an autoencoder to extract important features and learn the encoded representation of the time series. This approach helps to capture testing data points coming from a different population. We then train a prediction model based on this encoder's representation of data. An uncertainty estimation algorithm is used to estimate the model's uncertainty, which breaks it down to three different sources: model uncertainty, model misspecification, and inherent noise. To get the first two, a Monte Carlo dropout approach is used which is simple to implement and easy to scale. For the third part, a bootstrap approach that estimates the noise level via the residual sum of squares on validation data is used.

As a result, we could see that our proposed model can make a better prediction in comparison to our benchmarks. Although the difference is not big, yet it shows that making prediction based on encoding representation is more accurate. The anomaly detection results based on these predictions also show that our proposed model has a better performance than the benchmarks. This means that using autoencoder can improve both prediction and anomaly detection tasks. Additionally, we conclude that using deep neutral networks would show bigger improvement if the data has more complexity.

Sammanfattning

Målet med den här uppsatsen är att implentera ett verktyg för anomaliupptäckande med hjälp av LSTM autoencoders och applicera en ny metod för osäkerhetsestimering med hjälp av Bayesian Neural Networks (BNN) baserat på en artikel från Uber research group [1]. Pålitliga verktyg för att upptäcka anomalier och att göra precisa osäkerhetsestimeringar är kritiskt i många fält. På Telia kan ett sådant verktyg användas för många olika datadomäner, som i enhetsloggar för att upptäcka abnormalt beteende. Vår metod använder en autoencoder för att extrahera viktiga egenskaper och lära sig den kodade representationen av tidsserierna. Detta tillvägagångssätt hjälper till med att ta in testdatapunker som kommer in från olika grundmängder. Sedan tränas en förutsägelsemodell baserad på encoderns representation av datan. För att uppskatta modellens osäkerhet används en uppskattningsalgoritm som delar upp osäkerheten till tre olika källor. Dessa tre källor är: modellosäkerhet, felspecifierad model, och naturligt brus. För att få de första två används en Monte Carlo dropout approach som är lätt att implementera och enkel att skala. För den tredje delen används en enkel anfallsvikel som uppskattar brusnivån med hjälp av felkvadratsumman av valideringsdatan. Som ett resultat kunde vi se att vår föreslagna model kan göra bättre förutsägelser än våra benchmarks. Även om skillnaden inte är stor så visar det att att använda autoencoderrepresentation för att göra förutsägelser är mer noggrant. Resulaten för anomaliupptäckanden baserat på dessa förutsägelser visar också att vår föreslagna modell har bättre prestanda än benchmarken. Det betyder att användning av autoencoders kan förbättra både förutsägelser och anomaliupptäckande. Utöver det kan vi dra slutsatsen att användning av djupa neurala nätverk skulle visa en större förbättring om datan hade mer komplexitet.

Contents

Chapter 1

Introduction

1.1 Introduction

Anomaly detection in large companies and businesses has a vital importance and is considered as a complex task to address. Companies usually have defined Key Performance Indicators (KPIs) as metrics to understand how well their business is doing. Detecting anomalies in time can help them to save money and improve the quality of their services. Machine learning approaches can help to make automatic anomaly detection among a wide range of KPIs and help the companies to understand what is happening and what to expect to happen in future. An accurate model is required to learn patterns in data and detect the correct anomalies; otherwise, high rate of false positives or failures in detecting the anomalies can also lead to significant problems for businesses.

Many companies have a manual approach for detecting anomalies in different areas like their underlying infrastructure, various business applications, and business analysts. They usually have dashboards and people assigned to monitor daily or weekly reports of the operations or performance factors. In case something abnormal is seen, there are usually defined procedures to analyse the root causes. In this approach it is not possible to track all or most of the metrics at the same time and usually each metric will be monitored separately. Therefore, finding correlations and effect of each KPI on the others will be missed.

Another popular way is to define thresholds and generate alarms whenever a metric goes above or under the threshold. Finding the proper threshold for each metric or KPI needs a deep understanding of the KPI. Furthermore, the stasis of such threshold can lead to an increasing amount of false positive alarms, as well as a failure to detect anomalies. Consider an online retail

company as an example [2]. The company might see an unexpected increase demand for one product. The expectation is to see a raise in the revenue but surprisingly they see a drop instead of raise in the revenue. Monitoring these two metrics together can show that there is something abnormal happening and they need to start root cause analysis to investigate the problem. But without considering multiple metrics at the same time the company might not be able to identify the problem quickly and and thus lose money.

Defining what is abnormal and what is normal in data is a difficult question to answer. Depending on the type of data, its domain and history, the answer to this question varies. Time series anomaly detection algorithms are usually trained on normal data (without abnormal instances) to learn the normal pattern of a signal [3]. Then behaviour of an unseen data in future will be predicted. If it deviates from what is considered as normal, it will be detected as an unexpected pattern and will be reported as abnormal.

Due to the complex nature of anomaly detection, an automated and large scale anomaly detection tool can help businesses to prevent failures, saving money and resources, and create new business opportunities. Many researches are trying to apply machine learning and artificial intelligence methods to develop accurate algorithms for detecting anomalies. There are four types of machine learning methods: supervised, unsupervised, semi-supervised, and reinforcement learning. Using supervised methods requires labeled data. Annotating and labeling data for detecting anomalies in huge amounts of data and specially time series is not a practical task. Therefore, unsupervised, semi-supervised, and weakly supervised approaches are the main focus of the researches in anomaly detection problems. Reinforcement learning is about an agent that receives information about its environment and need to choose an action that will maximize some reward which is not related to this topic.

The goal of this thesis is to apply deep learning methods in an unsupervised way to detect anomalies in time series. Apart from skipping labeling the data, there is no need to do feature engineering using deep learning which is a complex task and required domain knowledge. Instead, the large number of parameters in deep neural network cells during training phase will be adapted to the model input history and they will learn the important features of the input data.

1.2 Definition of the problem

Outlier or anomaly detection is a broad subject with a large variety of application domains. Chandola, Banerjee, and Kumar [3], Gupta et al. [4], and many

others tried to provide an overview and classification for anomaly detection. Defining an anomaly by itself is a complicated problem [3] [5]. Depending on the domain and what angle we are looking at data, part of data can be abnormal or just a different trend, which is actually normal. When talking about anomalies, we need to be careful about what we call abnormal. Generally, anomalies are outliers in data that are different from usual distribution and normal frequency of data (w.r.t. most accurate representation of all the data). There are usually peaks in time series that look very different from the rest of data. But there are also rare and extreme events that are different from normal behaviour of data but are still considered normal. Figure 1.1 shows some examples of abnormal behaviour in time series.

Automatic time series anomaly detection is a very close concept to performing predictions in time series and plays an important role in it [6]. Predicting the future trend of a time series automatically is one of the most important applications of machine learning, especially for big organizations with a huge amounts of data. Since it is not easy to label such a large amount of data, unsupervised methods are used to find a solution to make predictions in time series. Automatic anomaly detection and prediction in time series has a variety of applications such as allocating resources in an effective way, health system monitoring, energy consumption, increasing the profit or income by better investments, fraud detection, predictive maintenance, etc. For a large enterprise like Telia with massive amount of system and social data, time series forecasting and detecting anomalies are of great importance. Due to the importance of detecting outliers in industry, many software packages are providing tools and packages for finding anomalies like R, SAS, etc [4]. In addition, one of the most important components of time series prediction is to provide a reliable prediction uncertainty as well.

In classical time series prediction methods, usually one model will be trained per each time series. In some cases these classical methods are combined with machine learning approaches, but they are still not easy to deal with for large scale data. An example of these classical methods is extreme value theory (EVT) [7] which is a branch of statistics. Univariate timeseries approaches are also considered as classical methods for time series predictions. Some common approaches for modeling univariate time series are autoregressive (AR) model, moving average (MA) model, and Frequency Based Methods [8].

Laptev et al. [9] and Zhu and Laptev [1] from Uber, proposed a novel end-to-end model for predicting number of ride requests that Uber receives every day. The goal was to develop an accurate prediction model for multiple time

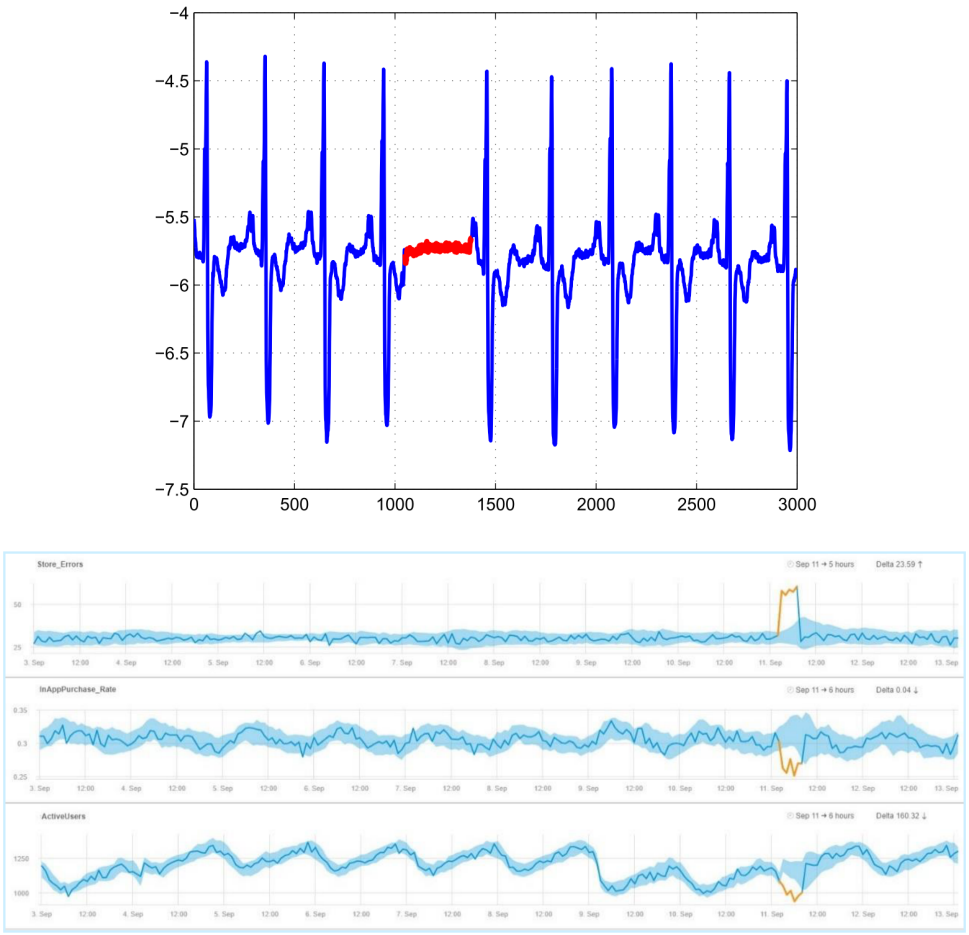


Figure 1.1: Examples of abnormal behaviour in time series. TOP: Human electrocardiogram [3]. The red part of the plot shows an abnormal heartbeat rhythm. BOTTOM: unexpected changes in multiple time series which are highlighted by orange color [2]

series that also takes into account many external factors effect on the prediction results. These external factors for Uber include the weather conditions like raining, windy days, temperature degree, etc. The challenge was to have an accurate prediction for special events (holidays, sport events, Christmas, New Year's Eve) that can also be different for different cities. Some of these factors happen rarely, sometimes only once in a year, and it is hard to predict them with classical time series models [9].

This thesis is based on the time series prediction and uncertainty estimation method proposed by Uber. Our goal at Telia is to implement this general time series prediction model and detect upcoming anomalies that can be applied on different domains. For example uncertainty estimation can be used in network monitoring systems with thousands of KPIs (key performance indicators) to automatically detect a failure; whenever the future values goes outside the 95% predictive interval an alarm is generated. To develop such a model for time-series predictions, they used Deep Recurrent Neural Network (RNN), more precisely Long Short Term Memory (LSTM) [10] networks, which will be described in detail in this report.

As part of our research, we are trying to take the advantage of RNN cells' large number of parameters and avoid doing extensive feature engineering while aiming for a high prediction accuracy. RNNs and LSTMs are promising models for processing time series due to their natural modeling of sequences[11]. Furthermore, we are trying to study if we can predict upcoming incidents with a long time (1-2 days) warning, based on the data from a large number of sources and from a long time period. Another question to answer is if we can in our predictions include estimates on quality of the predictions made.

1.3 Social Aspects

Using automatic anomaly detection algorithms instead of traditional way of monitoring thousands of metrics manually (using dashboards, raising alarms, etc), raise the typical concern about using artificial intelligence (AI) and replacing taking away jobs from people. On the other hand, to be able to deal with the scale of big data, it will not be possible to use manual approaches anymore. There are many controversial debates about how AI affects humans life style and replacing their jobs which are outside the scope of this report.

From a different perspective, using models to detect outliers in a huge amounts of data, can let us detect anomalies much faster than before. This is an impossible task for humans to look into and infer any meaningful infor-

mation from this amount of data. This can help society to react faster and be more agile. In these days when society is getting more and more complicated, agility, and re-activeness can be the key to many threats which may break the fabric of a healthy society.

Apart from helping companies with automation, anomaly detection can be used to predict urban crimes like burglary and robbery [12]. With an accurate prediction of a crime and prevent it from happening, the quality of life for citizens of a city can be improved.

1.4 Ethical Consideration

The data used for this project is only system generated data, not social data and contains no personal information, although the developed model can be used on time series of any other domain to detect outliers. Finding anomalies in data gives companies and organizations a chance to save money and create new business opportunities. One might think that all metrics in a business are related to money; this is not always true. Although most of those metrics indirectly affecting the revenue, detecting outliers can also save significant time and manpower that can be spent on other opportunities. Instead of creating a lot of dashboards and reports or setting thresholds manually, automatic anomaly detection can be used to make the process more reliable and easier.

Additionaly, we do see that getting access to the large amount of data to look for outliers needs to take privacy into consideration. Our work is not about how we should access these data. It is about after it has the access; then how to generate anomaly scores as fast as possible with high confident. Outlier detection models can be used to monitor ethical consideration and react to threats against them.

1.5 Sustainability

Society, large scale companies, huge institutions are examples of when a very small decision can have huge consequences. Today, everything is connected in ways which can be impossible to predict and know in advance e.g. health care, energy industry, social networks, and politics. Nowadays we need to monitor how our actions are impacting other connected entities. With a huge amounts of data, which should be monitored at scale and in a way to be able to measure its bias, a breed of models are needed to find outliers in our data. In these very

complex systems, we have no tool other than to monitor impact of our actions by checking it actively for any anomalies, and be able to react with much lower latency to make sure about the sustainability.

As mentioned before, general anomaly detection approach that can be applied on time series of different domains can help companies and organizations to save money and resources. One of the most important examples is the energy consumption problem. By monitoring machines and device behaviour, companies will be able to detect failures or other unusual pattern in machines, with no need of using a tool or assigning manpower to take care of them. Specially if the machines are located in a remote site, sending people for unnecessary reasons or false alarms will cost a lot. Furthermore, from the environment point of view, this will help to avoid unnecessary commutes which results in carbon footprint reduction. Finally, using an automatic monitoring system can help to have a more efficient management of energy consumption.

1.6 Outline

The rest of this report is organised as follows: chapter 2 describes the theoretical background of this project. It covers a summary of deep learning and unsupervised learning. It goes through the structure of Recurrent Neural networks (RNN) and Long Short Term Memory networks (LSTM), and describes LSTM implementation details in Keras. In addition, this chapter introduces Encoder-Decoder and Autoencoder models and a background about Bayesian Neural Networks (BNN). At the end, a short summary of similar works by other researchers is covered.

Chapter 3's focus is on the methods we used in this project. It starts with data preparation. Next, the design of baseline models and our proposed model will be explained. At the end of this chapter, prediction uncertainty and how we implemented it is described. These are all based on the paper from Uber [1]. The results of our experiments are presented in chapter 4 and future works is discussed in the last chapter.

Chapter 2

Theoretical Background

2.1 Deep Learning

Deep Learning is a sub-field of machine learning which is able to learn high-level representations of data in a supervised or unsupervised way [13]. It's a network of layers stacked on top of each other and their goal is to transform the input data into meaningful output. Each layer can be seen as a non-linear module that receives the output of the previous layer as its input. Deep Learning models learn to do this transformation automatically and that is one of the reasons that make them quite popular. Francois Chollet in [14] has a geometrical definition for neural networks: "a very complex geometric transformation in a high-dimensional space, implemented via a long series of simple steps." He simplifies this definition using an example of crumpling two papers with different colors into a ball. This paper ball illustrates the input data with two classes. If there were three papers with three different colors, there would be three classes in this dataset. Deep learning helps to find a way to transform this crumple ball back to two different classes of colors (two papers) again, see figure 2.1:

"With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time... [Deep Learning] takes the approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much the strategy a human would follow to uncrumple a paper ball."

Yoshua Bengio in his talk [15] describes Deep Learning as an algorithm that comes to the aid of beating the curse of dimensionality in data. He explains how the curse of dimensionality makes learning difficult in neural network:

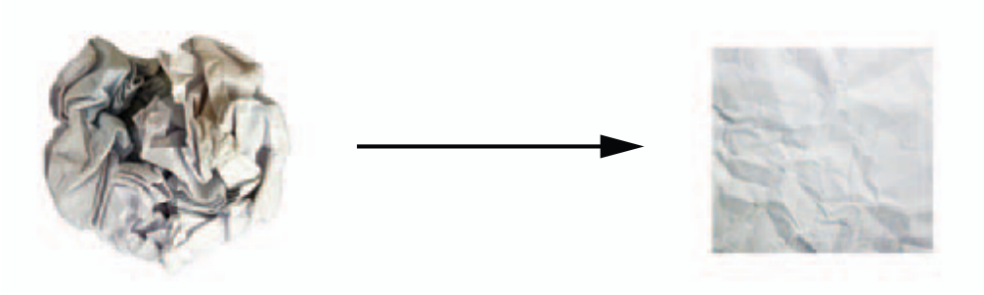


Figure 2.1: Uncrumpling a complicated manifold of data, picture adopted from [14]

"... how do we defeat the curse of dimensionality? In other words, if you don't assume much about the world, it's actually impossible to learn about it." He suggests that Deep Learning helps us to bypass the curse of dimensionality by making the model compositional, meaning by composing little pieces together. In other words by composing layers together and composing units on the same layer together, Deep Learning helps out to achieve that. He describes that deep learning tries to learn "feature hierarchies". Features of the higher level are formed by composing features from lower level, and that is the meaning of hierarchy.

The researchers in [16] [17] introduced the ability to extract features automatically, without a need to label the data, with Deep Learning. They show that an unsupervised pre-training method improves the performance and helps to have a more generalized model. This ability makes Deep Learning a good approach for anomaly detection since collecting labels has a lot of problems. Unsupervised learning with Deep Learning is expected to gain more attentions in the coming years [18] which will be discussed in more details in the rest of this report.

2.2 Unsupervised Learning and Representation Learning

Machine learning algorithms can be categorized in four main branches: Supervised learning, Unsupervised learning, Self-supervised learning, and Reinforcement learning [14]. The most common approach is supervised learning in which the algorithm tries to learn mapping input data (X) to output data (y). It is called supervised because we know the correct answers and when the algorithm goal is to approximate the output it will be corrected based on the true

value of (y). Classification, Regression, and Sequence generation are some of the supervised learning problems. Unlike supervised learning, unsupervised learning only gets the input data and has no clue of the correct answers. These algorithms are useful to find interesting structures in input data and usually used for denoising, compression, or finding correlations in the data. The third category of machine learning algorithms is called self-supervised learning and it sits in between of the two other categories. These algorithms don't need data to be labeled manually, instead, the labels will be generated from input data automatically. It is basically the data that provides supervision in this type of learning [19]. Autoencoders are one of the examples of self-supervised learning. The last category of algorithms, reinforcement learning, is an agent that receives information about its environment and need to choose an action that will maximize some reward. Our focus in this report is unsupervised and self-supervised learning.

Unsupervised learning has seen increased usage after the successful application of many different deep learning models, such as generative adversarial networks (GANs) [20], Long Short Term memory networks (LSTMs) [10] and variational autoencoder (VAE) [21]. The Canadian Institute for Advanced Research (CIFAR) is known as one of the pioneers of using unsupervised learning procedures for feature extraction.

One of the well known domains of unsupervised learning tasks is anomaly detection where there are no labeled data for training the network. Another issue is that the abnormal behavior in some datasets usually happens rarely and most of the data is normal points. Anomaly detection using unsupervised learning tries to learn the normal behavior of the data and learn the representation of training data with no anomalies. Therefore, any deviation from that normal behaviour will be considered as an anomaly. Authors in [21] used representation learning to automatically extract the features for video data. Videos and images are high dimensional structures which makes it very difficult to detect anomalies in them. Representation learning helps to automate feature extraction process while takes into account important prior information about the problem [22]. Representation learning is used in methods for reconstructing the input data, like Principal component analysis (PCA) and Autoencoders (AEs) [23].

2.3 RNN

Traditional neural networks don't have a memory that allows previously seen information for their current reasoning. Whereas, recurrent neural networks

(RNN) have loops in their architecture that allows keeping information from past. This loop passes the information from past steps to next step [24]. The internal memory of RNNs keeps information about input data in the form of weight matrices. To understand the structure of an RNN network, we start with a basic neural network that has only one hidden layer. It will transfer input vector X to output vector y as follow:

$$\mathbf{h}_t = \phi(W_{xh}\mathbf{x}_t) \tag{2.1}$$

$$\mathbf{y}_t = W_{hy}\mathbf{h}_t \tag{2.2}$$

where $\mathbf{x}$ is input to the network, W_{xh} is the weight matrix that connects inputs to the hidden layers, $\mathbf{h}_t$ is the output of a single neuron, W_{hy} is the weight matrix connecting the hidden layers to the output layer, and ϕ is an activation function like $\tanh$. Figure 2.2 shows an example of a basic neural network with four neurons and its W weight matrix. It shows how input maps to the hidden layer in a matrix. To simplify the notations, we are considering bias to be a column of x and h matrices. Figure 2.2 just shows input and first hidden layer of a neural network. There is also an output layer, $\mathbf{y}_t$, with its weight matrix, W_{hy}, that outputs the result of the network (eq 2.2).

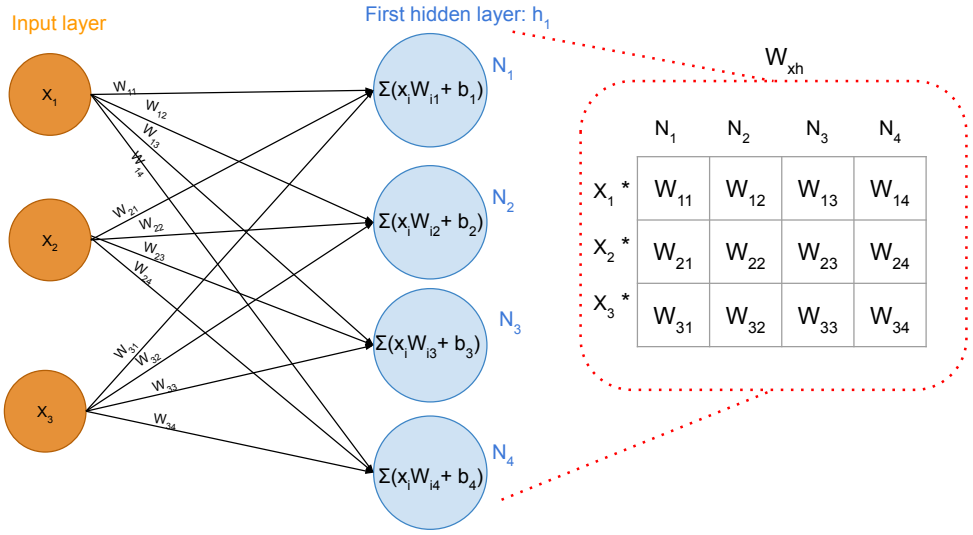


Figure 2.2: A basic neural network with its weight matrix, picture adapted from [25].

To enable memories in RNNs, the encoded information from one hidden layer will be sent as a memory from one timestep to the next one. The mathematical equation for it will be:

$$\mathbf{h}_t = \phi(W_{xh}\mathbf{x}_t + W_{hh}\mathbf{h}_{t-1}) \tag{2.3}$$

$$\mathbf{y}_t = W_{hy}\mathbf{h}_t \tag{2.4}$$

where $\mathbf{h}_t$ is a new state at each time step t and it will be passed to next timestep. $\mathbf{h}_{t-1}$ is the old state and our memory from past, $\mathbf{x}_t$ is input at timestep t, ϕ is an activation function. To get the new state $\mathbf{h}_t$ values, there are two weight matrices now: W_{hh} that has weights to move from one hidden state to another, and W_{xh} that contains inputs to hidden states weights. $\mathbf{y}_t$ in equation 2.4 is the output of the loop for each time step like t. The output layer has a weight matrix called W_{hy}. The unrolled graph in figure 2.3 shows the calculation process of RNNs. The weight matrix is the same at every step.

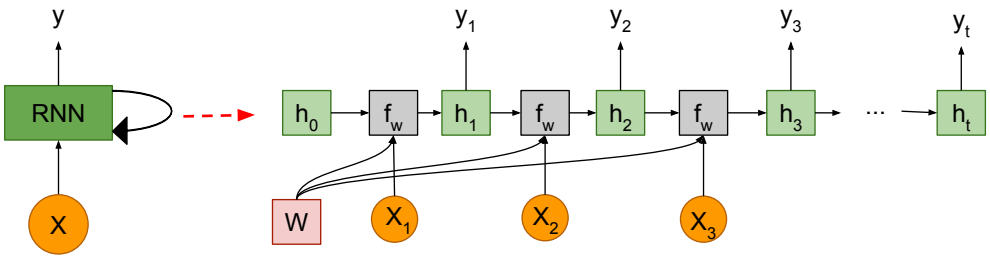


Figure 2.3: An unrolled recurrent neural network. Each green box is calculating $\mathbf{h}_t = F_W(\mathbf{h}_{t-1}, \mathbf{x}_t)$ which is a function with W as parameters as described in equation 2.3. The horizontal arrows are carrying $\mathbf{h}_{t-1}$ from previous timestep to the next one. Picture adopted from [26]

What was explained till now was the feedforward phase in RNN. After this phase the Loss will be calculated, and its gradient will be used in the backpropagation phase to corrected the weights and minimize the Loss. This should work in theory, but in practice it has been proven that RNNs are problematic to train [27]. That's because backpropagation of gradients will either explode or vanish after each step. The exploding issue can be solved by gradient clipping and the vanishing problem can be fixed by changing RNNs internal architecture. One of these architecture changes resulted in LSTM networks.

2.4 LSTM

Long Short Term Memory networks, LSTMs, were introduced to solve the issue with vanishing gradient in RNNs by changing their simple internal loop to a different structure. That makes LSTMs be capable of remembering long

periods of time. It can keep track of items' order in a sequence and learn the dependencies between them. Equation 2.5 lists the functions of an LSTM unit as described in [24].

In figure 2.4, hidden state $\mathbf{h}_t$, is the output of LSTM and is called LSTM capacity. The size of it will be chosen by the user. Apart from hidden state, LSTM also has an internal state called the cell state, c_t. In general, we don't use cell state as an output of LSTM unless there is a specific reason for it. Cell state is the main difference between RNN and LSTM internal components, which is like a memory for LSTM and keeps information from the past.

This information can be affected (added or removed) by LSTM gates. Each gate has a sigmoid function applied on their final results. This sigmoid function will generate a value between 0 and 1. First gate is "forget gate" that gets the previous state and new input values to decide how much of past information should be remembered and how much should be forgotten. The closer the sigmoid result is to 1, the more our LSTM unit remembers from past. In the same way, the closer the result is to 0, the less memory from past will be kept by LSTM unit. This result will affect cell state of the previous state, C_{t-1}. The second gate is "input gate" that decides how much of new information should be added to our previous knowledge. Similar to the forget gate, the sigmoid function is applied on the new input and previous state to make this decision. The result is multiplied with $\widetilde{C}_t$ to provide a new vector to be added to current cell state. The third gate in an LSTM unit is the "output gate" which decides on LSTM output and will affect h_t value. The sigmoid function in this gate works the same as what was described for the other gates.

$$
\begin{aligned}
forget\ gate &: f_t = \sigma(W_{xh_f} x_t + W_{hh_f} h_{t-1} + b_f) \\
input\ gate &: i_t = \sigma(W_{xh_i} x_t + W_{hh_i} h_{t-1} + b_i) \\
new\ input\ information &: \widetilde{C}_t = \tanh(W_{xh_c} x_t + W_{hh_c} h_{t-1} + b_c) \\
update\ cell\ state &: C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}_t \\
output\ gate &: o_t = \sigma(W_{xh_o} x_t + W_{hh_o} h_{t-1} + b_o) \\
hidden\ state/output &: h_t = o_t \tanh(C_t)
\end{aligned}
\tag{2.5}
$$

As equation 2.5 shows, there are three different weights for each gate: W_{xh}, W_{hh}, and $\mathbf{b}$. $\odot$ is element wise matrix product. Weights are matrices that represent a linear transformation from input to output. They will be calculated automatically based on the chosen shape of input and required output. Equation 2.6 lists the size of these weights. An LSTM layer with "h units" will have $4 * (hunits * hunits + hunits * n\_features + hunits * 1)$ parameters.

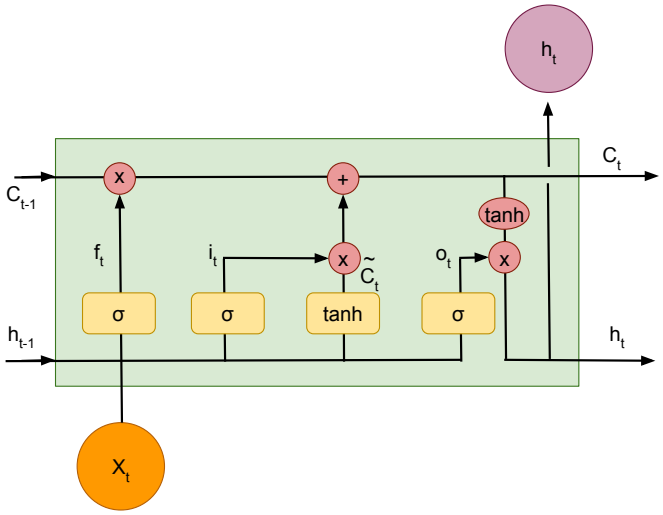


Figure 2.4: LSTM structure. Picture adopted from [24]

$$
\begin{aligned}
W_{xh_f} \in \mathbb{R}^{hunits * n\_feat},\ W_{hh_f} \in \mathbb{R}^{hunits * hunits},\ b_f \in \mathbb{R}^{hunits * 1} \\
W_{xh_i} \in \mathbb{R}^{hunits * n\_feat},\ W_{hh_i} \in \mathbb{R}^{hunits * hunits},\ b_i \in \mathbb{R}^{hunits * 1} \\
W_{xh_c} \in \mathbb{R}^{hunits * n\_feat},\ W_{hh_c} \in \mathbb{R}^{hunits * hunits},\ b_c \in \mathbb{R}^{hunits * 1} \\
W_{xh_o} \in \mathbb{R}^{hunits * n\_feat},\ W_{hh_o} \in \mathbb{R}^{hunits * hunits},\ b_o \in \mathbb{R}^{hunits * 1}
\end{aligned} \tag{2.6}
$$

Hidden state h_t and cell state C_t vectors will be vectors with shape of $\mathbb{R}^{hunits * 1}$. For example, a single layer of LSTM with two neurons, 3 inputs of dimension 1, will have the following weight matrices:

$$
\begin{gathered}
4 * [W_{xh} \in \mathbb{R}^{2 * 1},\ W_{hh} \in \mathbb{R}^{2 * 2},\ b \in \mathbb{R}^{2 * 1}] \\
h_t, c_t \in \mathbb{R}^{2 * 1}
\end{gathered}
$$

2.5 LSTM in Keras

Keras[1] is an open source python library that provides high level APIs for neural networks. It is built on top of Tensorflow[2], CNTK [3], or Theano[4]. To imple-

[1] https://keras.io/
[2] https://www.tensorflow.org/
[3] https://docs.microsoft.com/en-us/cognitive-toolkit/
[4] http://deeplearning.net/software/theano/

ment and run experiments for this project we used Keras on top of Tensorflow. In comparison to Tensorflow, Keras is user friendly. For this project we are using version 2.2.4 of Keras. To be able to get the required results, one needs to have a good understanding on how these APIs are implemented in Keras. Particularly with LSTM Neural Networks there are important details that a developer needs to have a good understanding of. The following sections will explain the implementation of some of these details.

2.5.1 Input shape

Data in Keras is stored in a multi-dimensional matrix, a Numpy array, called *tensor*. LSTMs input must be a 3-dimensional tensor that represents time sequence order and has the shape of `(n_samples, timesteps, n_features)` [14], as shown in figure 2.5. `n_samples`: a sequence of inputs that has overlap with the next sequence is one sample. `timesteps` or `lookback`: is the number of times that the LSTM should be unfolded and is what we know as a neuron. `n_features`: one feature is one observation at a time step. This is more clear in figure 2.6 and 2.7. Figure 2.6 shows an input with four timesteps and one feature. Therefore, the LSTM has been unfolded four times (one sequence). In figure 2.7, the number of timesteps is three and there are five features in each sequence. It worth mentioning that in Keras functional API, the input layer itself is not a layer, but only a tensor that will be sent to an LSTM layer.

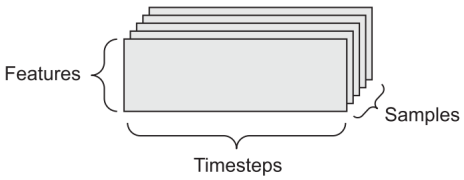


Figure 2.5: A 3D time series data tensor. Picture adopted from [14]

2.5.2 Units

This is the number of hidden units. Based on Keras documentations this will define the dimension of output. It will set the size of hidden state and cell state matrices in LSTM. As mentioned before, this will be considered as the LSTMs capacity; therefore, the bigger the number of units is the more learning

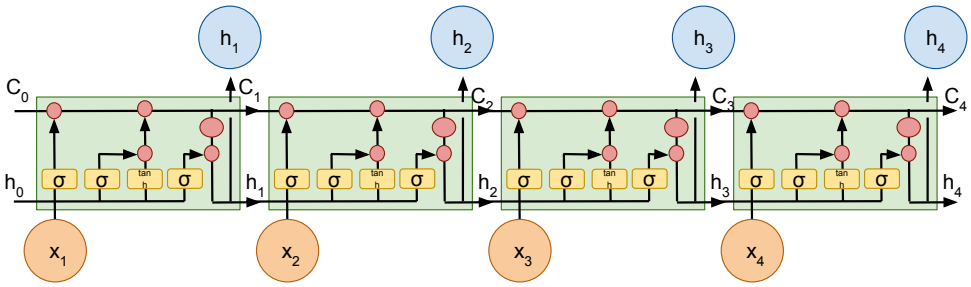


Figure 2.6: An LSTM layer with input shape of $(batch\_size, 4, 1)$

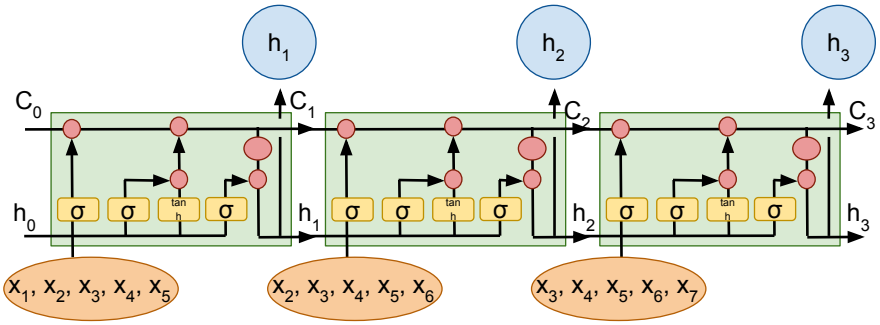


Figure 2.7: An LSTM layer with input shape of $(batch\_size, 3, 5)$

capacity the LSTM has. This is one of the parameters that needs to be tuned in order to prevent overfitting during the training phase. h_t and in some cases C_t are the outputs of an LSTM layer and number of units will specify their dimension. If the `return_state` option of the LSTM layer is set be True, then C_t will be returned as output beside h_t. This will be explained in more detail in next section.

2.5.3 return_sequences

Depending on the model we are developing, LSTM can have different output approaches. As a side note, one should consider that the hidden states are the outputs of an LSTM layer. In Keras, LSTM layer has an option called `return_sequences`. By default this option is set to `False`, meaning that the output is only the results of last LSTM hidden state or last timestep of the current sequence. Setting this option to `True` will tell LSTM to return all hidden states from all timesteps in the sequence (not only the last one). Figure 2.8 illustrates the results of setting `return_sequences` to `True`. For example if the input shape is `(batch_size, timestep =28, n_features=1)`

and `LSTM(unites=32, return_sequences=True)`, the output will have be in 3D shape of `(batch_size, timesteps=28, units=32)`. Otherwise, if the LSTM layer is defined without setting `return_sequences` the output will be `(batch_size, units=32)`.

For this project, to create an autoencoder model using LSTMs, we need to connect LSTM layers together. As mentioned before, LSTM input shape must be a 3D tensor. There are two approaches to achieve this in Keras: setting `return_sequences` option to `True` or using a `RepeatVector()` layer in between of two LSTM layers. Figure 2.9 illustrates how `RepeatVector()` works. It will copy the last hidden state of the last timestep as an input to next LSTM layer. More explanation on how we used these two options for our model is given in the method chapter.

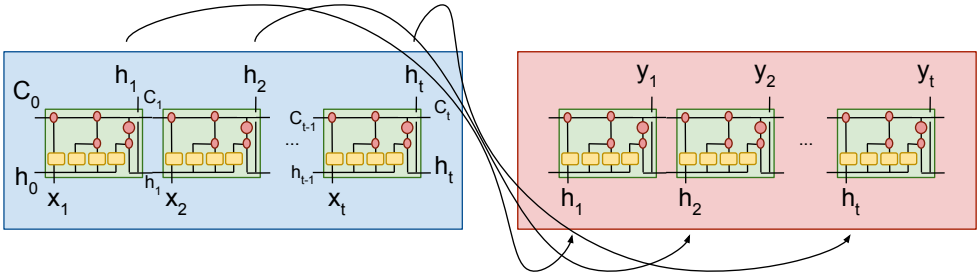


Figure 2.8: Connecting two LSTM layers using `return_sequences = True`

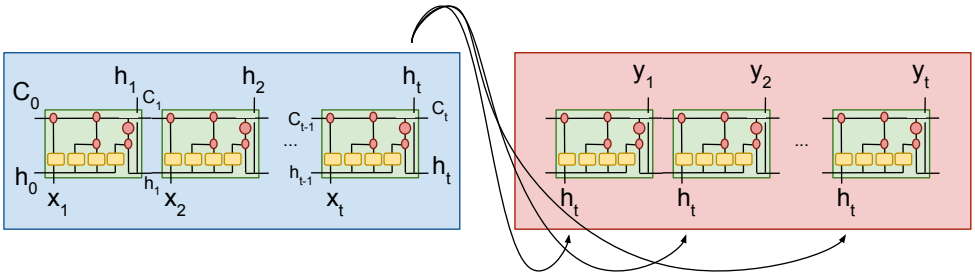


Figure 2.9: Connecting two LSTM layers using `RepeatVector()`.

2.5.4 return_state

A related point to consider is that in addition to the hidden states that can be output of LSTM, there is the cell state. Keras implementation of LSTM has a boolean option, `return_state`, which returns cell state (c_t) at last time

step if it is set to be `True`. To be more precise, if `return_state` option is set to `True` it will return three values:

- The hidden state for last time step: h_t
- The hidden state for last time step: h_t (again)
- The cell state for last time step: c_t [28]

2.5.5 Stateful

Keras documentation for this LSTM option is a bit unclear and misleading. Statefulness in LSTM is related to using batch in training process. Using `batch_size` means how many samples the network should see before updating the weights. By default, LSTM in Keras will reset the cell state after each batch. If `stateful` is `True`, the cell state of the last timestep of i_{th} sample will not be saved for next sample $(i + 1)$. Instead, the final cell state of i_{th} sample from current batch will initialize cell state of i_{th} sample of the next batch. Figure 2.10 shows an example of how LSTM maintains states between two batches if `stateful` option set to be `True`. A more accurate description is samples in a batch are independent from each other and the default behaviour is to keep cell state only among each sample's timesteps [29], not between samples of the same batch. There is one state per each sample in a batch and after each batch, by default, all of these states will be reset for the next batch. That is a common misunderstanding of how maintaining state works in LSTM in Keras. That is also the reason that one should not use `shuffle=True` while using stateful LSTM. After visiting all training data (all batches) at the end of one epoch, `model.reset_states()` will be called to reset all states and start over.

2.5.6 Dropout and recurrent_dropout

In order to prevent overfitting during the training process of neural networks, one can use dropout regularization. Using dropout will randomly set the output of some hidden units of a layer to zero during training. This works quite well for a feedforward or Dense layer. But it needs a more complicated approach for recurrent neural networks like LSTM. Yarin Gal in [30] explains how dropout should be used in an RNN layer. In short, the same dropout pattern should be used for all timesteps in an RNN layer. Keras implementation of LSTM has already added variational dropout and provides two options for it: `dropout` and `recurrent_dropout`. `dropout` option is the dropout

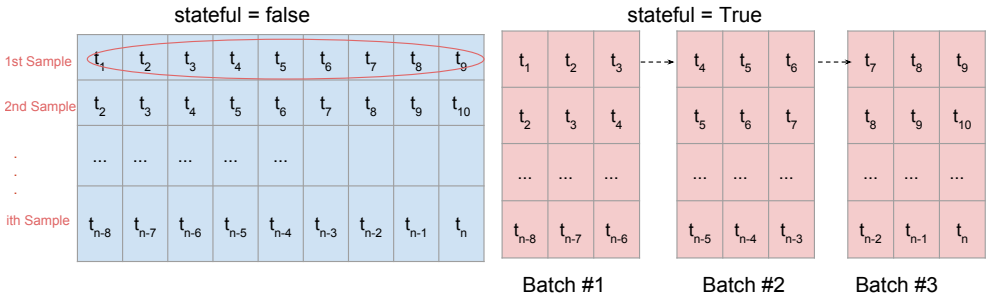
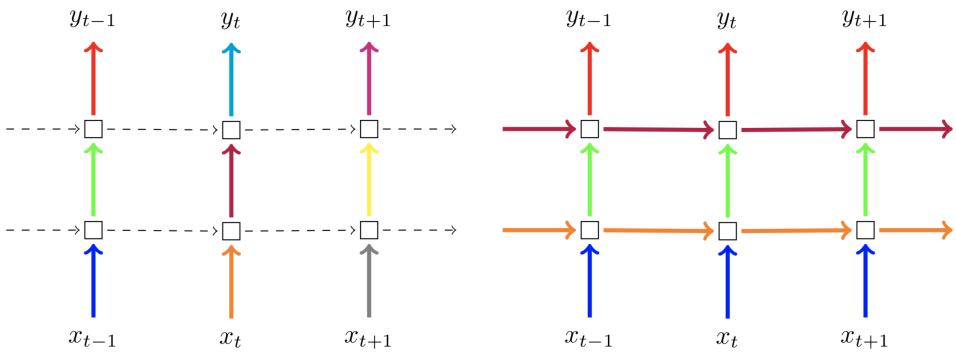


Figure 2.10: The blue box on the left, `stateful` is `False`; therefor, there is only one batch containing all samples. On the right, for the red boxes `stateful` is `True`, meaning that you will pass a long sequence divided into smaller pieces or batches. The cell state of the last timestep of i_{th} sample from current batch, will be passed to i_{th} sample of next batch to initialize its value.

rate for input units of the layer (W_{xh}). Keras documentation describes this option as: "Fraction of the units to drop for the linear transformation of the inputs". `recurrent_dropout` option belongs to dropout rate between the recurrent units of the layer (W_{hh}). Keras documentation describes this option as: " Fraction of the units to drop for the linear transformation of the recurrent state". Figure 2.11 shows the difference between the two dropout technique implementations.



(a) Naive dropout RNN (b) Variational RNN

Figure 2.11: Left: the standard dropout technique, Right: bayesian dropout , picture from [31]. Different colors mean different dropout masks.

In our project, to implement Monte Carlo dropout in Keras, we used `training` option for some of the layers. By default, Keras won't use dropout during prediction. Using this option will keep the recurrent dropout running in the

forward pass and will use the same dropout rate during test phase:

```
denseLayer = Dense(h_units)(inputLayer)
drLayer = Dropout(drRate)(denseLayer, training=True)
```

It should be mentioned that the effect of dropout on RNN units has been studied by different groups and some have reported that it does not always improve the results as expected [32].

2.6 Autoencoders

One of the most common unsupervised leanings methods for automatic feature extraction is encoder-decoder [33]. An encoder-decoder model is composed of two models: an encoder model and a decoder model. The encoder model transforms input data to a latent space. The decoder maps this output of encoder to another desired space. One instance of encoder-decoder is autoencoder that uses the same idea of having two models, encoder and decoder, to reconstruct the input data. An autoencoder is a specific design of neural networks that tries to learn a representation of its input.

Depending on the design of the model, there are different types of autoencoders available like variational [34], Sparse, and Denoising autoencoders [35]. No matter the type of autoencoders, they are usually used for extracting useful information and features from input data in a unsupervised way. There are other applications for autoencoders as well, such as hashing, sequence to sequence learning, data compression, and data generation [36]. To be more precise, autoencoders are considered as self-supervised models in machine learning, that is because they already have some information of what their output should look like.

The structure of the autoencoder usually has a symmetric design (but they don't have to be symmetric); number and size of layers in encoder is the same as decoder but in reverse order. They both share the encoding layer which is encoder's output and decoder's input. Figure 2.12 shows the general structure of an autoencoder.

The general equation for a basic autoencoder with one layer (feed forward) is shown in 2.7 and 2.8 equations. In these equations, function $f()$ represents encoder model and function $g()$ represents the decoder model. The whole model, $gof(x)$ is the reconstruction of input x. We call $\mathbf{h}$ the encoded representation of input $\mathbf{x}$. σ_1 and σ_2 are activation functions, $W^{(1)}$ and $W^{(2)}$ are weight matrices and $\mathbf{b}^{(1)}$ and $\mathbf{b}^{(2)}$ are bias vectors. Equation 2.8 shows decoder part of the autoencoder that maps $\mathbf{h}$ to the reconstruction $\widetilde{\mathbf{x}}$.

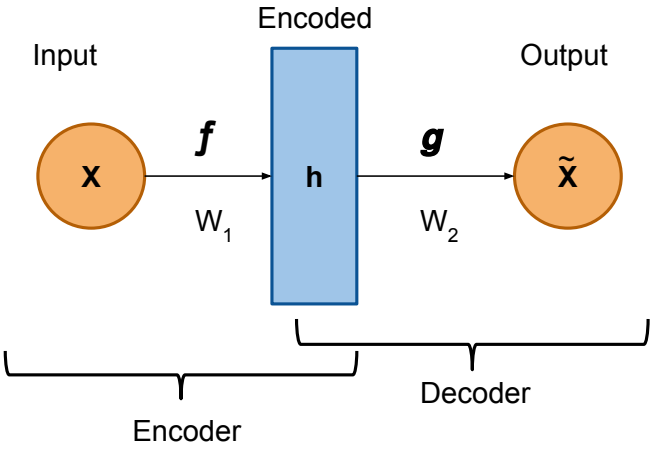


Figure 2.12: General structure of Autoencoders

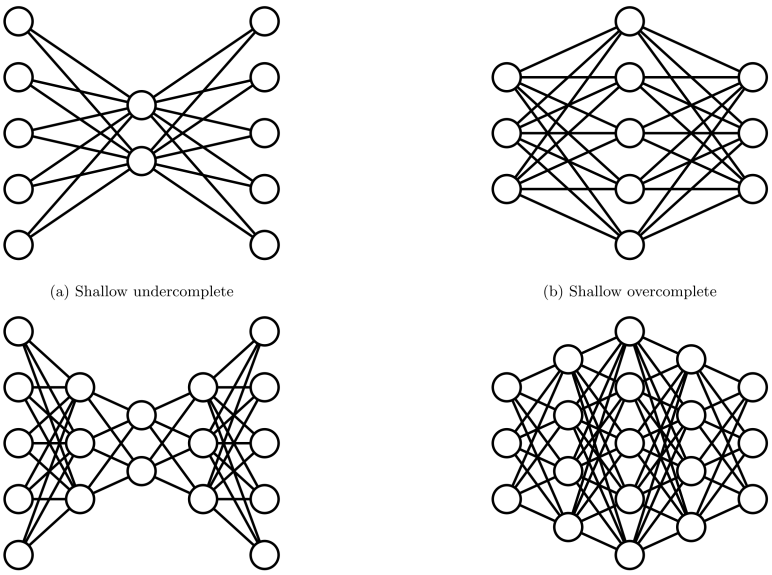
$$\mathbf{h} = f(\mathbf{x}) = \sigma_1(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \tag{2.7}$$

$$\widetilde{\mathbf{x}} = g(\mathbf{h}) = \sigma_2(W^{(2)}\mathbf{h} + \mathbf{b}^{(2)}) \tag{2.8}$$

The Loss function for autoencoders, defined in equation 2.9, aims to minimize the reconstruction error [37].

$$\mathcal{L}(\mathbf{x}, \widetilde{\mathbf{x}}) = \|\mathbf{x} - \widetilde{\mathbf{x}}\|^2 = \|\mathbf{x} - \sigma_2(W^{(2)}(\sigma_1(W^{(1)}\mathbf{x} + \mathbf{b}^{(\mathbf{1})})) + \mathbf{b}^{(\mathbf{2})})\|^2 \tag{2.9}$$

Based on their architecture, Charte et al. [36] categorised autoencoders into four different types. First they categorize them based on the dimensionality of the encoding layer to Undercomplete and Overcomplete. Undercomplete is an autoencoder that its encoded layer has a lower dimensionality than the input to the autoencoder. If the encoded layer has higher dimension than input, then it is called Overcomplete. This type of autoencoder needs to apply more restrictions to avoid copying input to output. [36] also categorise autoencoders based on the number of layers in them. As a result, there are two types of autoencoders: Shallow and Deep. A Shallow autoencoder has three layers: input, encoding (one hidden layer), and output. A Deep autoencoder however has more than one hidden layer. A combination of these two categorization gives us four types of autoencoder shown by figure 2.13.



(a) Shallow undercomplete (b) Shallow overcomplete

(c) Deep undercomplete (d) Deep overcomplete

Figure 2.13: Different types of autoencoder structure. Picture adopted from [36]

One can create an autoencoder model based on LSTM networks for sequential data. The capability to remember order sequences of input in LSTMs, makes LSTM autoencoder capable of learning useful information of the ordering of the sequential input. After training the autoencoder model, one can only use the trained encoder part of it to encode input to its embedded representation.

2.7 Bayesian Neural Networks

One of the big challenges for time series anomaly detection is that it usually depends on many external factors that need to be considered to have a reliable and accurate prediction. When the uncertainty of the model is required, Bayesian neural networks can be helpful. The authors of the paper from Uber [1] are taking advantage of a Bayesian LSTM model to do nonlinear feature extraction [38] [11]. They also provide an uncertainty estimation for time series prediction to show how much the prediction is accurate and trustable.

In short, from a statistical point of view, training an standard neural network is equivalent to Maximum Likelihood Estimation (MLE) to estimate

networks parameters (weights and bias). But this method leads to overfitting problem in neural networks. Regularization is one solution for this issue but it is not the perfect solution. Using regularization turns the MLE estimation to a maximum a posteriori probability (MAP) estimation which considers priors for weights. But the method still has problems from a statistics point of view. Instead, Bayesian neural networks were used to change the standard neural networks MLE optimisation to "posterior inference". The rest of this section will explain more about how Bayesian networks work.

In standard neural networks, the parameters (weights and biases) have fixed values, while in Bayesian neural network, each parameter has a probability distribution. The output of a Bayesian neural network can be a set of outputs (from running the network multiple times) that each represents a realization of the parameter distribution and an uncertainty can be defined for each of them. Figure 2.14 shows the difference between how parameters are defined in these two type of neural networks.

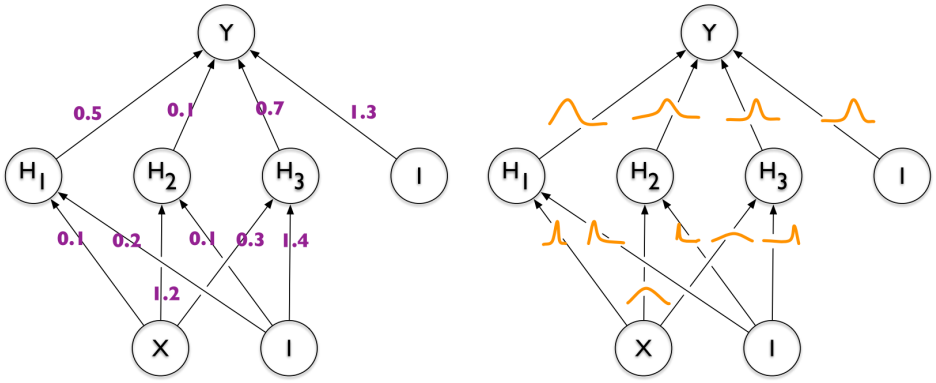


Figure 2.14: A standard neural network assigns fixed values to weights and biases (left), while in a Bayesian neural network, each parameter has a probability distribution (right). Picture adopted from [39].

Considering Bayes rule in equation 2.10, $P(W|D)$ is the posterior distribution that we need to find weights and biases for it. In this equation W denotes neural network parameters and D denotes data. $P(W)$ is the prior distribution for the parameters and $P(D|W)$ is the likelihood which represents the neural network. To calculate $P(D)$ one needs to solve equation 2.11 which is a very difficult task; instead, an approximation method like variational inference [40] can be used. In other words, we will try to find the closest probability distribution to posterior instead of calculating the equation above. Apart from

variational inference, there are other approximation inference approaches like Markov chain Monte Carlo (MCMC) [41] or Laplace's method [42] too, but they either have very poor results or are very slow for such a big amount of parameters in a deep neural network.

$$P(W|D) = \frac{P(D|W)P(W)}{P(D)} \tag{2.10}$$

$$P(D) = \sum_{j} P(D|W_j)P(W_j) \tag{2.11}$$

There are many new approaches that are trying to use variational inference for posterior approximation. [43] provides an algorithm based on stochastic optimization for optimization of the variational lower bound. [39] achieved extensive results introducing an algorithm called Backprop that uses variational inference. Authors of the Uber paper in [1] used a Monte Carlo dropout (MC dropout) framework to estimate model uncertainty which is based on [44] and [31]. In this project we tried to follow the same framework proposed by Uber to make a reliable uncertainty estimation for time series at Telia.

2.8 Related works

Due to the importance of the anomaly detection problem, a lot of research has been done on this topic to study different techniques and domain of application for anomaly detection. The definition of anomaly and outlier, the related techniques and applications for different type of temporal data including time series is thoroughly studied in a survey by Gupta et al. [4]. Chandola, Banerjee, and Kumar [3] provide a review on different aspects of anomaly detection problem. They classified the techniques for anomaly detection to the following categories: Neural Networks-Based, Bayesian Networks-Based, Support Vector Machines-Based, and Rule-Based. [45], [46], and [47] also reviewed anomaly detection techniques in detail. [48] classifies anomaly detection solution to three approaches: statistical approaches, neural network based approaches, and nearest neighbor based approaches. [49] and [50] studied anomaly detection in time series data. Authors of [51] provide a comprehensive survey on anomaly detection methods based on deep learning technology. They provide cutting edge technologies for anomaly detection in their underlying approach and their application domain. Figure 2.15 shows their contribution in classifying these techniques.

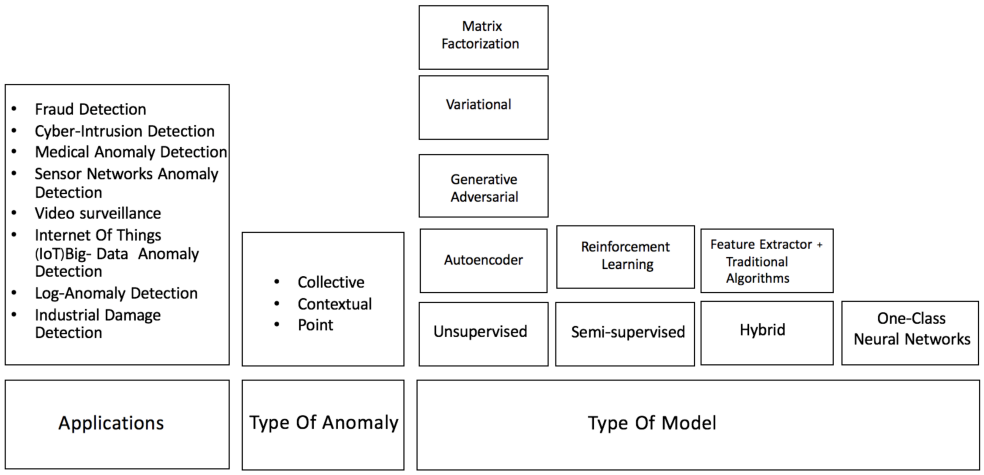


Figure 2.15: Deep learning methods for anomaly detection classification. Picture adopted from [51]

In their paper, Yu et al. [52], classify the common approaches for time series forecasting and anomaly detection to auto-regressive moving average (ARMA) [53], state space models such as hidden Markov model (HMM) [54], and deep neural networks. [52] used a novel method of neural networks called Tensor-Train RNN (TT-RNN) for multivariate forecasting in time series. This new architecture learns nonlinear features and their higher order correlations and uses tensor decompositions to compression of number of parameters.

Deep learning's ability to extract higher order features makes this technology a popular approach for capturing patterns within and across time series [55]. Specially the LSTM networks help to do time series prediction and anomaly detection with less human effort. One particular type of time series anomaly detection that is becoming more popular recently is extreme event prediction. In classical time series forecasting models, there is one model per each time series (e.g [56]). Although these models usually have accurate results, they are not flexible and not easy to scale .

In the first paper from Uber [9], the authors used LSTM Autoencoder for automatic feature learning and developed an scalable end-to-end model for rare-event prediction. They provide a general model to assign drivers in an efficient way during special days at Uber. They claim that their approach will train a generalized model based on data from some cities and can make predictions for all cities. Apart from the data related to number of requested rides in a day, they also added additional features like wind speed, temperature, and

precipitation. Their model consists of two LSTM models. First an LSTM autoencoder extracts useful features from input data. The output is features vectors that will be concatenated with the new input and will be fed to the second model which is a stacked LSTM model that makes the prediction. However, the input to the second model is not clearly described in the paper. To estimate forecast uncertainty, the authors used bootstrap method. They first estimate the model uncertainty based on the autoencoder results and then get the prediction uncertainty based on the prediction model, and repeat it 100 times. Finally, they combine these two as the whole model uncertainty estimation. The most interesting part of their published results is where the trained model on the Uber data was applied to the M3-Competition dataset and surprisingly they get reasonable results.

In their second paper from Uber [1], a similar approach with Bayesian deep learning model is provided for time series prediction and Monte Carlo dropout to get uncertainty estimates of deep neural networks. The proposed model in the first paper was generating a lot of false positives in its results. This issue was addressed in their second paper [1] by adding information about uncertainty using Bayesian Neural Network. [1] describes that the prediction uncertainty comes from three different sources: Model uncertainty, Model misspecification and Inherent noise. The details of this method and how to define these uncertainties are explained in third chapter.

Traffic volume and changes prediction for near future using deep neural networks and LSTMs have become an interesting problem for researchers, and various models were developed and studied. Bike flow prediction [57], forecasting of power demand [58], taxi demand prediction [59], hourly demand of bike-sharing using graph convolutional neural network [60], real-time prediction of taxi demand using LSTM-MDN (Mixture Density Networks) learning model [61], traffic prediction in the roads using Deep Ensemble Stacked Long Short Term Memory (DE-SLSTM) [62], etc. A good summary about traffic prediction researches is provided by [63] and [64].

Flunkert, Salinas, and Gasthaus [6] from Amazon also used encoder-decoder model and developed DeepAR for probabilistic time series forecasting. They mainly focused on multivariate time series forcasting. DeepAR is a seq2seq model that works based on Autoregressive recurrent network to learn the normal behaviour of all time series. It uses Monte Carlo sampling during prediction time to provide probabilistic prediction estimation.

Using deep neural network, Huang et al. [12] developed DeepCrime, a crime prediction framework, to detect urban crime patterns and predict them before they happen. They used attention mechanism in hierarchical recurrent

network for time series prediction. A category dependency encoder captures the complex interactions between regions and categories of occurred crimes in a latent space. Then a hierarchical recurrent framework with attention mechanism was developed to capture the dynamic crime patterns. They presented the results of an experiment on NYC data which showed a significant improvement of prediction accuracy in compare to their baseline model.

In [65] a collective anomaly detection based on LSTM networks is used for an intrusion detection in a network system. They trained an LSTM model based on normal data, then performed prediction for each timestep. The prediction error of a collection of timesteps will be considered to detect an anomaly.

A group from Numenta [66] proposed an unsupervised anomaly detection algorithm based on Hierarchical Temporal Memory (HTM). HTM is not a machine learning algorithm [67] but it is a learning algorithm with the ability to learn high order sequences.

Another interesting and successful usage of LSTM networks for detecting anomalies in a complex system is presented in [68]. Due to the lack of labeled data, they used an unsupervised or semi-supervised approach based on LSTM for spacecraft monitoring systems for multivariate time series data at NASA. To deal with large amount of telemetry channels that generates data in their systems, they trained single LSTM prediction model for each of the channels independently. They also proposed a novel dynamic error thresholding approach. This approach however had a high rate of false positive alarms; to address the issue they used a pruning procedure for anomalies.

Marchi et al. [69] used a denoising autoencoder in combination with LSTM network to detect abnormal acoustic signals. Anomaly detection in their work was actually an acoustic novelty detection that tries to identify novel (abnormal) acoustic signals that are different from their training data.

There have been many works on using time series anomaly detection techniques in medical data to recognize problems related to human health . [70] propose a predictive model to detect arrhythmia problems of human heart in Electrocardiography (ECG) signals. They developed an LSTM neural network to detect normal or abnormal behaviours in ECG data. [71] also tried to use Back Propagation Network (BPN), Feed Forward Network (FFN) and Multilayered Perceptron (MLP) on ECG data to classify it to two abnormal and normal classes.

Many studies have used an encoder-decoder model to learn a representation of the input data in an unsupervised way. [72] used an LSTM encoder-decoder model to learn representations of video sequences to show how the embedded layer learns and extracts features. Videos are high dimensional data

and they used encoder-decoder model to learn the video representation. They feed the video as a sequence of frames to an encoder and get the representation as its output. For the decoder model they took three approaches: A decoder to reconstruct the input video in a opposite direction, a decoder that can predict the future frame, and a combination of these two decoders. Figure 2.16 illustrates the model. The reconstruction decoder tries to learn important features of input and forget minor details to be able to reconstruct the input. The predictor decoder needs to remember more information about the recent seen frames to be able to do an accurate prediction and therefore, it tends to forget about the more distant past. Using the combination decoder, the encoder model needs to find a trade off between learning important features and the most recent sequences. Therefore, the model will not be able to copy the input to output. It can not forget all the past and old learnings either. In this model design, the decoder has this option to either receive the last generated output frame as input which is called a "conditioned decoder" or it will not receive this information which will be named an "unconditional decoder". In figure 2.16 it is shown as dotted boxes.

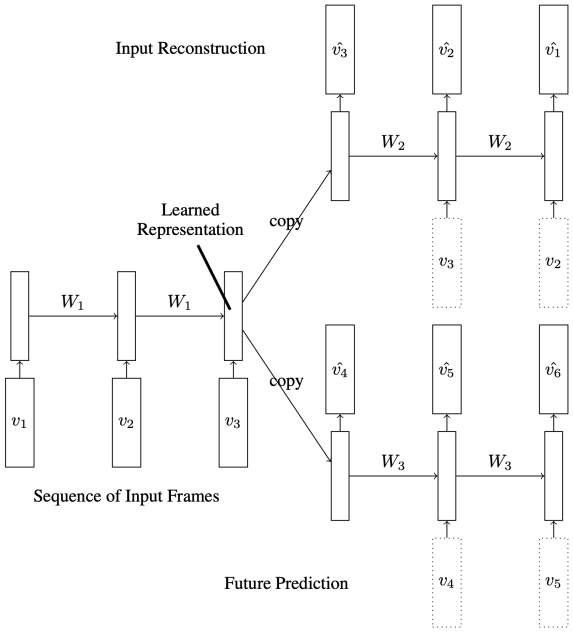


Figure 2.16: An LSTM encoder-decoder model with a combination of two decoders. Picture adopted from [72]

Chapter 3

Methods

3.1 Dataset

For this project, we got data for one of the network KPIs in Telia that shows the number of successfully established calls per day for each antenna in BTS towers. This data is a daily log of this KPI for a year and half. The goal is to predict the next day's number of calls based on the last 28 days values and predict if there will be an abnormal change in the number of calls. For these experiments, we selected time series with no missing values, this is the time series that have all the 548 days values and that left us with 138 time series. Figure 3.1 shows three examples out of 138 time series we have. Each time series belongs to the number of successfully established calls in a day for one antenna. Each time series has a different behaviour from the others depending on the location of the antenna. We consider the sudden changes (increases and drops) in the time series patterns as abnormal behaviour and that is what we aim to predict.

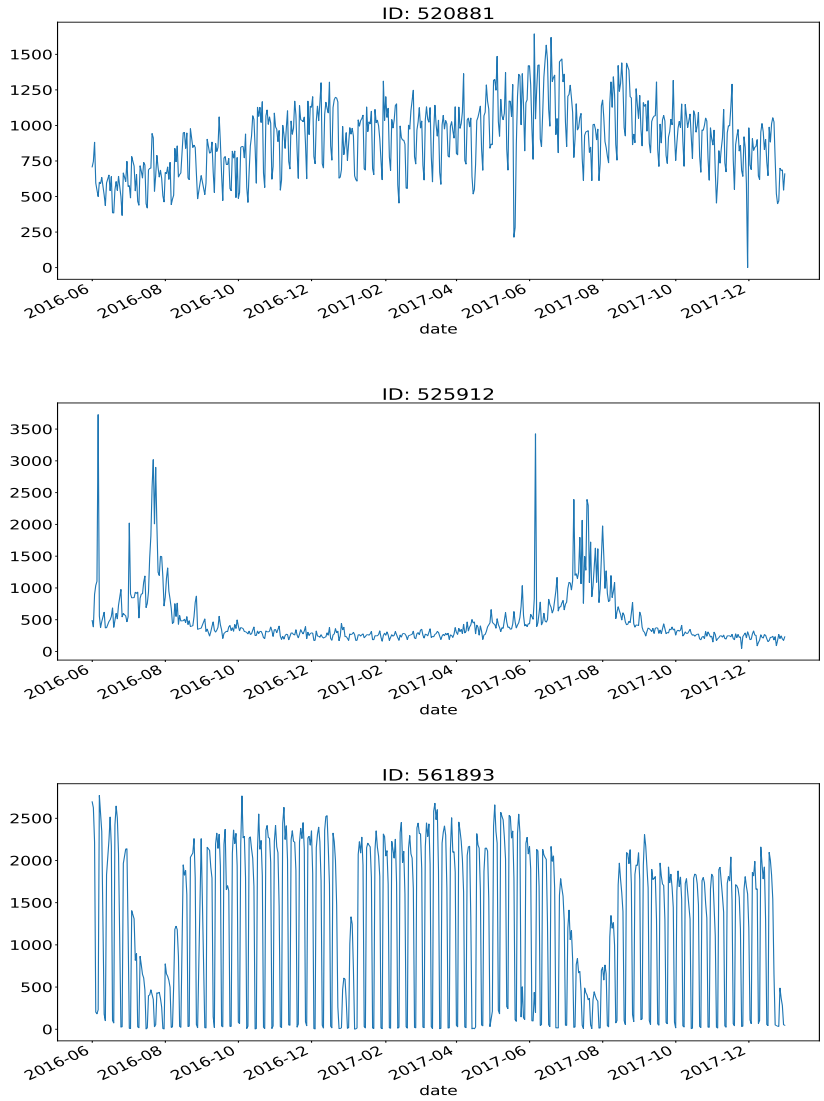


Figure 3.1: Examples of raw data signals

As figure 3.1 shows, each of these time series are in different scales. Therefore, one of the data preprocessing steps for us was to transform all time series to be in the same scale. In general, data preprocessing makes the raw data ready to be fed to the neural network. Usually, data preprocessing phase includes normalization, handling missing values, vectorization. For time series, the usual preparations are Power Transform, Difference Transform, Standardization, and Normalization [73]. Power transform technique is used to make data look more like a normal (Gaussian) distribution. Difference Transform,

also known as de-trending, is used to remove for example a seasonal structure from a time series. Standardization (also called z-score) is used to transform data to have a mean of zero and standard deviation of 1, equation 3.1. Finally, normalization transform data to for example a scale of 0 to 1, or -1 to 1 (min-max scaler), equation 3.2. Also, it is important to mention that in a machine learning problem, it is always required to have an inverse transform to rescale the prediction results to their original scale.

$$Z_x = \frac{x_i - \overline{x}}{\sigma} \tag{3.1}$$

$$MinMax_x = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{3.2}$$

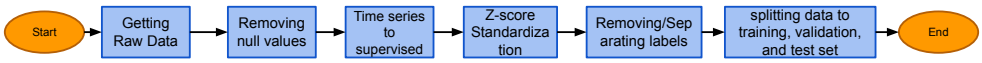


Figure 3.2: Data preprocess

The flowchart in figure 3.2 shows the steps we took to prepare the data. As mentioned earlier, the first step is to remove time series with missing values. The next step is generating the timesteps which is called "time series to supervised". By supervised we mean each 28 timesteps are followed by the label that we are supposed to predict (which is the 29th day's number of calls). We chose window size of 28 days as number of timesteps and the prediction is for one day ahead (29th day). Also, we tried two different approaches to generate these supervised timesteps: one-feature approach and a multi-features approach. In one-feature prediction illustrated by figure 3.3, each time series is concatenated one after the other. In case of multi-features model, each time series is considered as one feature itself, figure 3.4. Since there are 138 selected time series from the whole dataset, it gives us 138 features. We put the first timestep of all 138 time series at first positions, then the second timestep of all time series go after them and so on.

The next step in the flowchart is scaling time series. At first we were using min-max scaling to scale all time series between 0 and 1. We also tried the min-max scaling between -1 and 1. But for both of these scaling we ran into an issue for model training which was that model was not converging. Therefore, we changed the scaling to z-score standardization. One reason could be that some time series behave quite different from the others and they have big variations in their values. Using min-max scaler, we were basically ignoring these differences which was wrong. As described earlier, z-scores transform

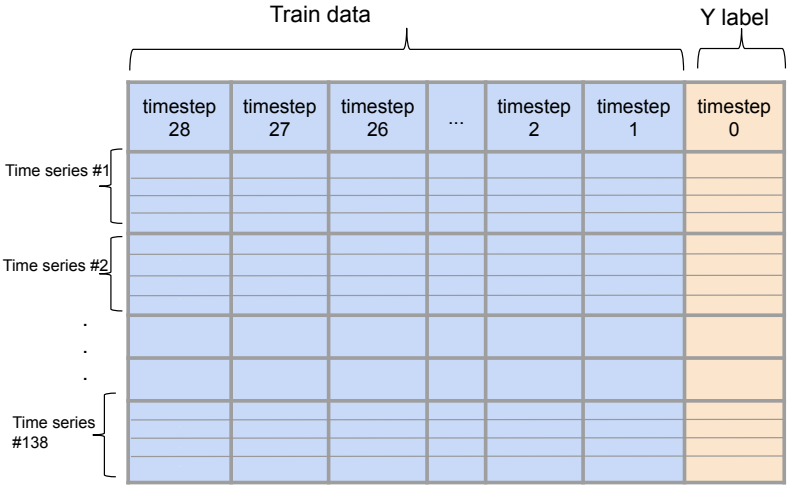


Figure 3.3: One feature approach: Converting time series to a supervised structure while considering all time series as one feature. Each time series is the data related to one antenna.

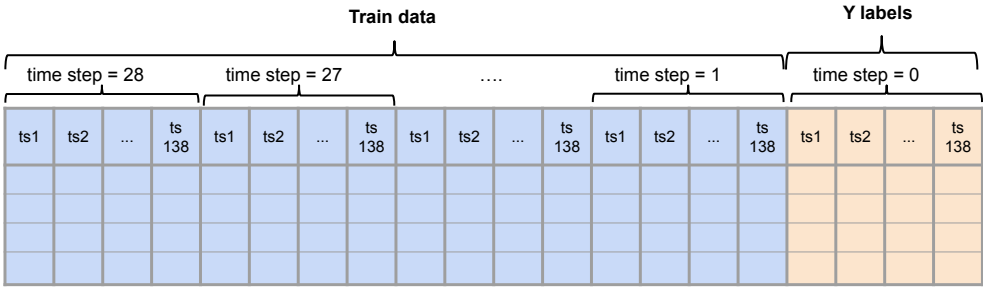


Figure 3.4: Multi feature approach: Converting time series to a supervised structure while considering each time series as an independent feature. The train data starts with timestep 28 of all time series, then time step 27 of all 138 time series till time step 1 of all 138 time series. The Y labels are the future time step of all time series. Each time series (ts) is the data related to one antenna.

raw data values to have a mean of zero and standard deviation of one. They consider the differences in score between each time series and make our time series have a more homogeneous pattern.

We took two different approaches for standardization one-feature and multi-features case. For one-feature case, each time series is scaled independently from the others, meaning there is one scaler for each time series and the $\overline{x}$ in equation 3.1 is calculated separately for each time series. While for multi-features case, the 138 time series are scaled all together. The $\overline{x}$ in equation 3.1 was calculated for all the 138 time series.

Before the last step in data preprocess flowchart, we need to remove the labels from our data and make it unsupervised. The data includes the 'Y' labels, they will be removed to generate unsupervised data and later these labels will be used for calculating the model residual. At the end, the data is split into 3 sets: 2/3 of data, which is 12 month from june 2016 to june 2017, is used for training; from the last 6 month, 4 months of data was assigned for validation and the last 2 months for test data set. We split their related 'Y' labels in the same way.

Figure 3.5 shows a sample of final data which is standardized and split to three datasets of training, validation and test.

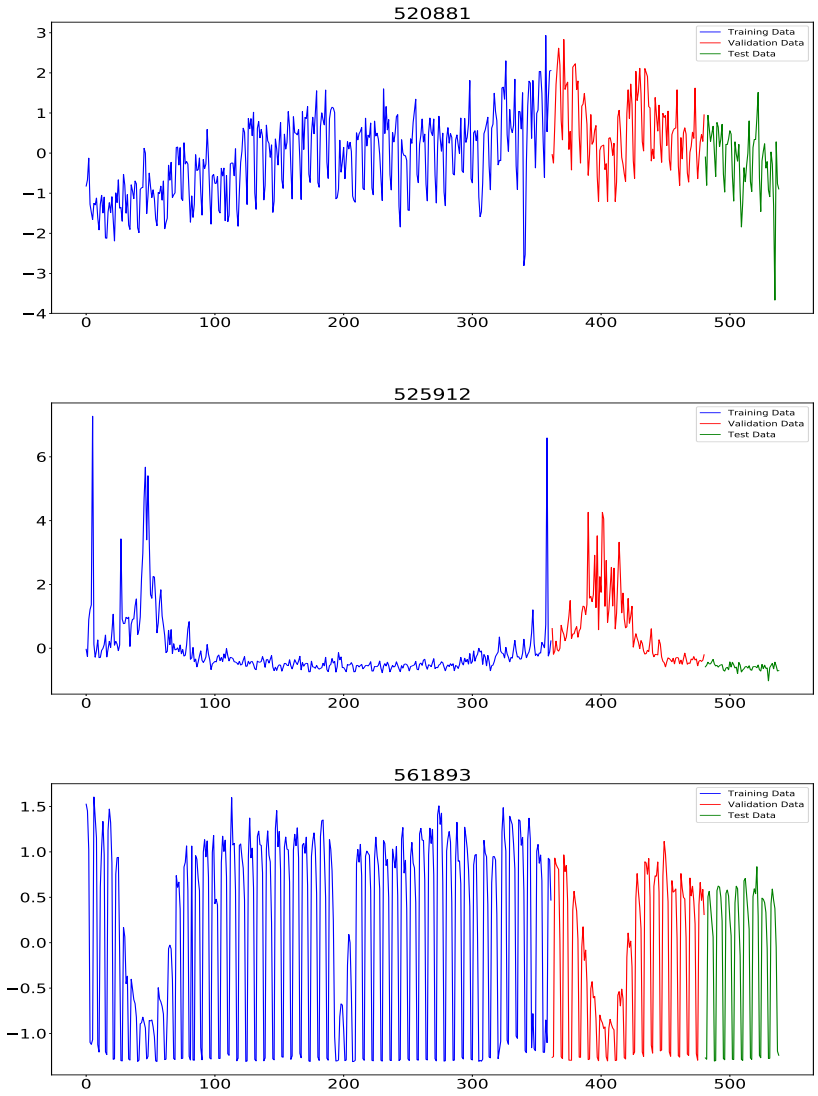


Figure 3.5: Zscore normalized data examples. Each time series is splitted to train data (blue), validation data (red), and test data (green)

3.2 Model Design

3.2.1 Baseline model #1: MLP

As our first baseline, we defined two Multilayer Perceptron (MLP) networks. One is an MLP model with one feature and the second one is an MLP model with multi features. As described in section 3.1, our dataset could be consid-

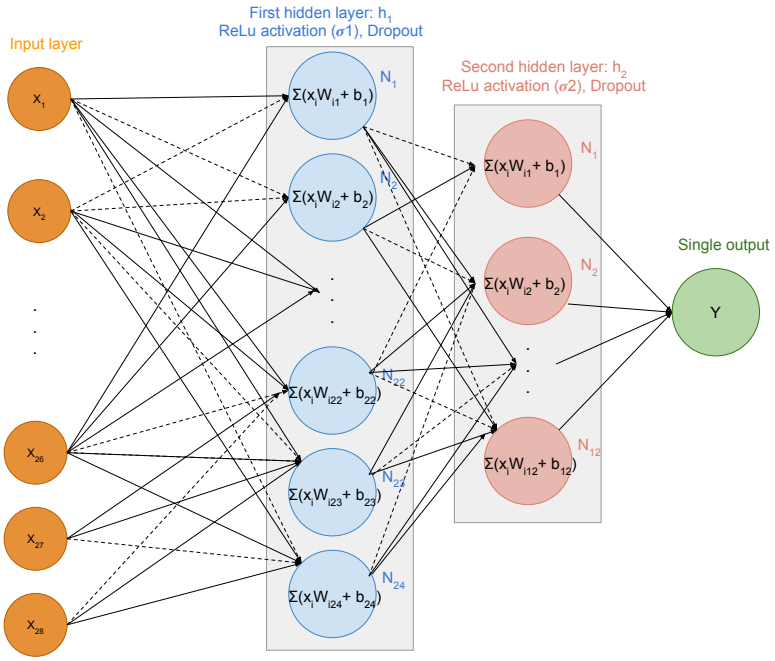


Figure 3.6: MLP model with one feature.

ered in two ways: to consider all time series as one feature only, or to treat each time series as one independent feature. Therefore, we need to have two MLP models for each of these datasets. Each model has two Dense() hidden layers and one Dense() layer with one unit as output layer.

Figure 3.6 shows the model structure for MLP model with one feature. The implementation code for this model is shown by listing 3.1. Each neuron in one layer is connected to all the neurons of the next layer. The input shape is (batch_size, 28 * 1) where 28 is our sequence length and input dimension, and 1 is number of features. First hidden layer has 24 neurons. Therefore, the output for this layer is (batch_size, 24). The second hidden layer has 12 neurons and its output has the shape of (batch_size, 12). The last layer which is the output has only one neuron since our goal is to predict one step ahead. The model will receive values for the past 28 days and predicts the value for day 29. Figure 3.7 illustrates the weight matrices for this model (bias matrices are not shown for simplicity).

Listing 3.1: MLP Model with one feature in Keras

```
seq_len = 28
n_features = 1
dropout_rate = 0.1
```

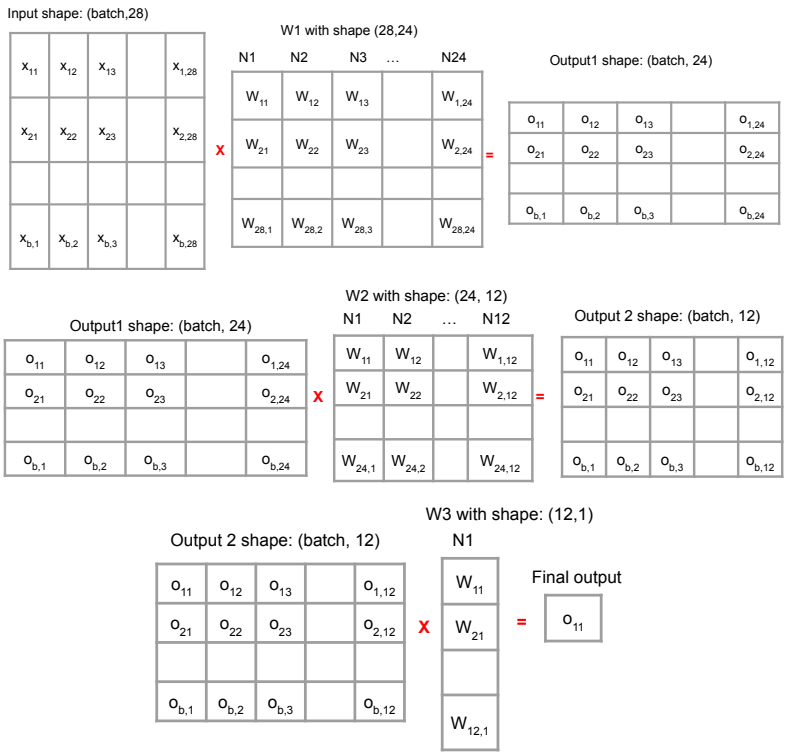


Figure 3.7: Weight matrices for MLP model with one feature and two hidden layers. The bias matrices are not shown for the sake of simplicity.

```
input_layer = Input(batch_shape=(batch_size, seq_len * n_features))
first_layer = Dense(24, input_dim=28, activation='relu')(input_layer)
if dropout_rate:
    first_layer = Dropout(dropout_rate)(first_layer)
second_layer = Dense(12, activation='relu')(first_layer)
if dropout_rate:
    second_layer = Dropout(dropout_rate)(second_layer)
out = Dense(1)(second_layer)
model = Model(input_layer, out)
```

Equation 3.3 shows the computation of the whole network. o_n^3 denotes the output of the third dense layer which has n=1 neuron (Y value in figure 3.6), that means the output for each 28 input is only one value (the goal of the model is to predict one step in future after seeing the value of the last 28 days). In this equation, x_i is the input to the model, $w_u^l v$ denotes the connection between v:th neuron in layer l-1 to the u:th neuron in layer l, b_u^l is the bias of the u:th neuron in layer l. σ_1 is the activation function in the first dense layer and σ_2 is the activation function for the second layer. We did not use activation for the

output layer.

$$o^3_{n=1} = \underbrace{\Bigg[\sum_m w^3_{nm}\Bigg[\overbrace{\sigma_2\Big(\sum_j w^2_{kj}\underbrace{[\sigma_1(\sum_i w^1_{ji}\mathbf{x}_i + \mathbf{b}^1_j)]}_{\text{First Dense Layer}} + \mathbf{b}^2_k\Big)}^{\text{Second Dense Layer}}\Bigg] + \mathbf{b}^3_n\Bigg]}_{\text{Third Dense layer}} \quad (3.3)$$

We used Mean squared error (MSE) as Loss function for this model. Equation 3.4 shows how to calculate this Loss where y is the predicted value, y' is the ground truth label, and N is the number of observations. The Loss function shows the error in the predicted value compared to the expected result. In other words, the Loss function shows the distance between the model output and the desired output.

$$\jmath = \sum_{i=1}^{N} \frac{1}{N}(y_i - y'_i)^2 \quad (3.4)$$

To update the weights and improve the results of the network, backpropagation phase is used. The goal is to minimize the error that Loss function is giving us. This is done by calculating the gradient, i.e. the partial derivate of Loss function $\jmath$ with respect to the weights elements, w_i and b_i. Let's call the different parameters of the network that affect the Loss function θ. The gradient descent of the Loss function $\jmath$ with respect to these parameters will be $\frac{\partial \jmath(\theta)}{\partial \theta}$. It shows how much the parameter θ, for example weight w_i, changes in the Loss and it will be calculated for each layer using the chain rule. Equation 3.5 shows how to update θ parameters using the gradient descent method. In this equation, γ is a scaler hyperparameter called learning rate that updates the parameters.

$$\theta = \theta - \gamma \frac{\partial \jmath(\theta)}{\partial \theta} \quad (3.5)$$

In order to speed up the process of learning and minimizing the Loss function, specially when the network is big and the size of the training data is large, we use optimizer algorithms. Stochastic Gradient Descent(SGD) is a variant of gradient descent that is very common for this purpose. Instead of updating parameters for the Loss calculation on the whole dataset, the SGD algorithm divides the dataset into batches and updates parameters for each batch Loss calculation. There are other optimizer algorithms such as RMSprop, AdaGrad, and Adam. Each of them have their own parameters to be tuned, which

we call them hyperparameters. For our experiments we mainly used Adam and RMSprop.

As mentioned earlier, we also tried the MLP model with multi features where each time series is considered as a feature. We used the same architecture as MLP with one feature for this model too. The model has two hidden layers and it will be similar to figure 3.6. The difference is in the shape of input and output layers and the first and last weight matrices. The input for 138 features (number of total time series available) and 28 timesteps will be in the shape of $(batch\_size, 28 * 138) = (batch\_size, 3864)$. The weight matrix W_1 will have shape of (3864, 24) since it has 24 neurons (same as MLP with one feature). W_2 will be the same size, $(24, 12)$, since the second layer will have 12 neurons. The output layer will output the predicted value for timestep 29 for each time series. Therefore, it will have the output shape of $(batch\_size, 138)$. The weight matrix for output layer, W_3, has the shape of $(12, 138)$.

3.2.2 Baseline model #2: vanilla LSTM for prediction

As our second baseline, we used a vanilla LSTM model with two LSTM layers. The input to this model is in the shape of (batch_size, 28, 1), where 28 is the number of time steps and 1 is the number of features. We tried this model with different numbers of hidden units: 28, 30, 50, and 128 for the first LSTM layer and 7, 10, 15, and 20 for the second LSTM layer. The output layer is a Dense layer with one hidden unit to predict the value for the day 29 based on information from the last 28 days. The math behind an LSTM cell and its gates is described in equation 2.5, and we used MSE as Loss function. Also, we tried both Adam and RMSPops as optimizers and compared the results of them. We also tried tanh and relu as activation function for the LSTM layer.

Listing 3.2: Vanilla LSTM Model in Keras

```
seq_len = 28
n_features = 1
dropout = 0.1
recurrent_dropout = 0.1
output_dropout = 0.1

input_layer = Input(batch_shape=(batch_size, seq_len, n_features))
first_layer = LSTM(50
                , activation='relu'
                , dropout=dropout
                , recurrent_dropout=recurrent_dropout
                , return_sequences=True
                , stateful=True)(input_layer)
if output_dropout:
    first_layer = Dropout(output_dropout)(first_layer)
```

```
second_layer = LSTM(20
                , activation='relu'
                , dropout=dropout
                , recurrent_dropout=recurrent_dropout
                , return_sequences=False
                , return_state=False
                , stateful=True)(first_layer)
out_layer = Dense(n_features)(second_layer)
model = Model(input_layer, out_layer)
```

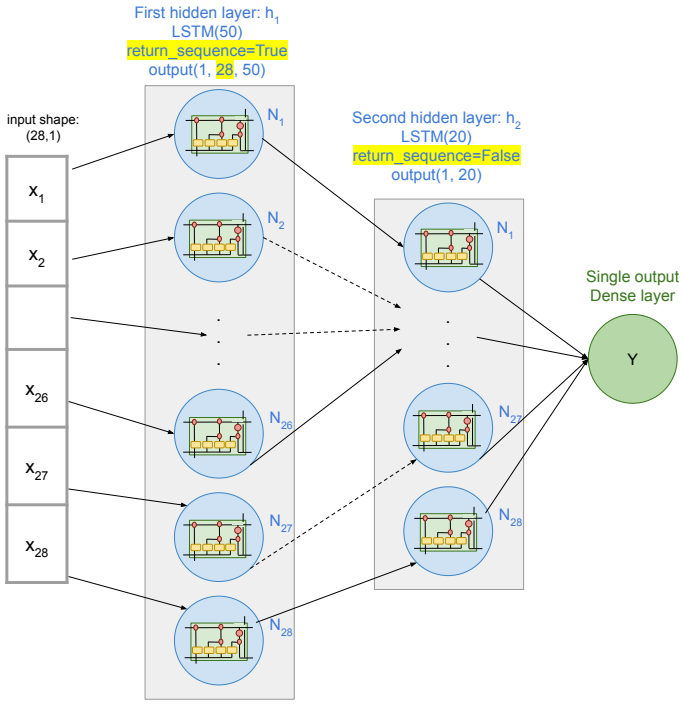


Figure 3.8: Vanilla LSTM model. Dashed lines illustrate dropout on the output of the layer. Both LSTM layers are unfolded 28 times. The input shape for first LSTM layer is (n_samples, 28, 1) and for second LSTM the input is the output of previous layer, which has the shape of (n_samples, 28, 50)

.

The model design is illustrated in figure 3.8 and the implementation code is listed in 3.2. To understand how data goes through these LSTM layers and how it changes we show only one instance of the input data with shape of (1, 28, 1) in figure 3.8 and follow its transformation through the network. Since the number of time steps is 28 our first LSTM layer, `LSTM(50)` will be unfolded 28 times. That explains $(N_1, N_2, ..., N_{27}, N_{28})$ of first hidden layer in the picture. The weight matrices for this layer will have the following size:

$$W_{xh} \in \mathbb{R}^{50 * 1},\ W_{hh} \in \mathbb{R}^{50 * 50},\ b \in \mathbb{R}^{50 * 1}$$

and since there are 4 of each matrices for each LSTM gate the final size of these matrices will be: $W_{xh} \in \mathbb{R}^{200 * 1}$, $W_{hh} \in \mathbb{R}^{200 * 50}$, $b \in \mathbb{R}^{020 * 1}$. As the code for this model in 3.2 shows, this layer has `return_sequences` option set to true; therefor, the output of this layer will have a 3D shape of (1, 28, 50). The second layer is a `Dropout()` layer which will be applied on the output of the first LSTM layer and will randomly set some of the weights to zero. This is shown in the picture with dashed lines. Then, there is the another LSTM layer, `LSTM(20)`, that receives the output of the dropout layer as its input. Based on its input shape, the LSTM units of this layer should be unfolded 28 times. The weight matrices will be in the following size:

$$W_{xh} \in \mathbb{R}^{20 * 50},\ W_{hh} \in \mathbb{R}^{20 * 20},\ b \in \mathbb{R}^{20 * 1}$$

and in total for the four gates the weight matrices will be: $W_{xh} \in \mathbb{R}^{80 * 50}$, $W_{hh} \in \mathbb{R}^{80 * 20}$, $b \in \mathbb{R}^{80 * 1}$. The output of this layer will have a 2D shape of (1, 20).

The last layer of this model is a fully connected layer, `dense(1)`, that makes the 29th timestep prediction. This layer has two weight matrices, W and b, with size of (1, 20). The output will be one scaler which is a predicted value for timestep 29th based on the last 28 timestep values.

3.2.3 Our proposed model: AE + prediction

Our proposed model based on the Uber paper consists of two parts: a reconstruction autoencoder and a prediction model.

3.2.3.1 AE reconstruction

To generate a representation of our input time series, an Autoencoder with reconstruction goal was implemented. It gets a 3D input data with shape (batch_size, 28, 1) and outputs a reconstruction of its input. Listing 3.3 is the code for this model in Keras.

Listing 3.3: Autoencoder Model in Keras

```
from keras.layers import Input, Dropout, LSTM
timesteps = 28
n_features = 1
dropout = 0.1
recurrent_dropout = 0.1
output_dropout = 0.1
```

```
input_enc = Input(batch_shape=(batch_size, timesteps, n_features))
first_enc = LSTM(32, activation='relu'
                 , return_sequences=True
                 , dropout=dropout
                 , recurrent_dropout=recurrent_dropout
                 , stateful=True)(input_enc, training=True)
if output_dropout:
    first_enc = Dropout(output_dropout)(first_enc, training=True)
encoded = LSTM(7, activation='relu'
                 , return_sequences=False
                 , dropout=dropout
                 , recurrent_dropout=recurrent_dropout
                 , stateful=True)(first_enc, training=True)
decoder_input = RepeatVector(timesteps)(encoded)
first_dec = LSTM(32, return_sequences=True)(decoder_input, training=True)
out_decoder = LSTM(n_features, return_sequences=True)(first_dec,
    training=True)
autoencoder_model = Model(input_enc, out_decoder)
```

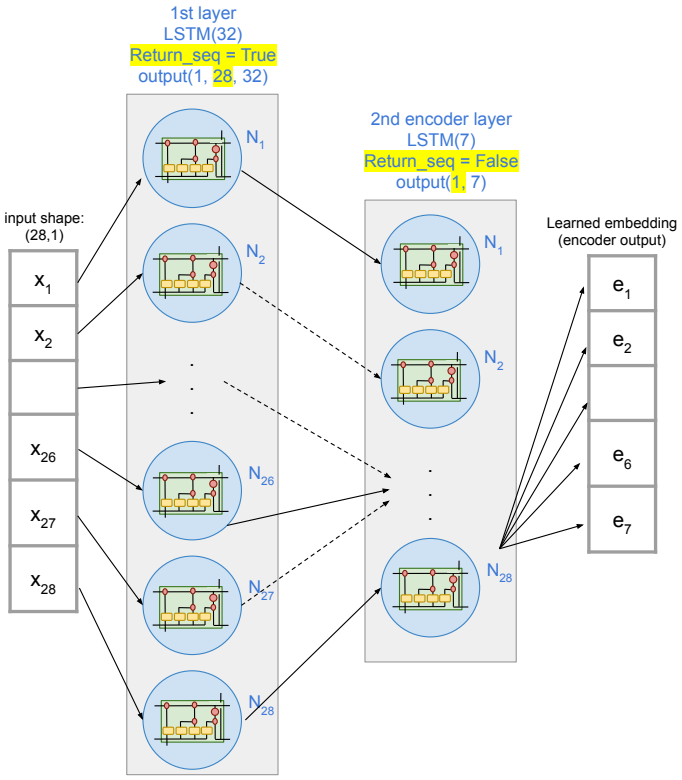


Figure 3.9: Autoencoder: Encoder model

Figure 3.9 illustrates the encoder part of the model and how does the data goes through its layers. In this picture, only one instance of input data with a shape of (1, 28, 1) is shown where 28 is the number of time steps and one

is number of features. This means that the first LSTM layer, `LSTM(32)` in the code, should be unfolded 28 times. The weight matrices for this layer will have the following size:

$$W_{xh} \in \mathbb{R}^{32*1},\ W_{hh} \in \mathbb{R}^{32*32},\ b \in \mathbb{R}^{32*1}$$

Since there are 4 of each of these matrices for each LSTM gate, the final size of these matrices will be: $W_{xh} \in \mathbb{R}^{128*1}$, $W_{hh} \in \mathbb{R}^{128*32}$, and $b \in \mathbb{R}^{128*1}$. The output of first LSTM layer is in 3D shape of (1, 28, 32) because for this layer we set the `return_sequences` option to `True`. Therefore, the output of the first LSTM layer will be in shape of 3D and each cell of the next layer will receive the output of all the cells of this first layer.

The second layer of encoder is `LSTM(7)`. It gets the output of previous layer as its input which is in the shape of (1, 28, 32). As a result, this LSTM layer will be unfolded 28 times and the weight matrices will be as follow:

$$W_{xh} \in \mathbb{R}^{8*28},\ W_{hh} \in \mathbb{R}^{7*7},\ b \in \mathbb{R}^{7*1}$$

Since there are 4 of each matrices for each LSTM gate, the final of the matrices will be: $W_{xh} \in \mathbb{R}^{32*28}$, $W_{hh} \in \mathbb{R}^{28*28}$, and $b \in \mathbb{R}^{28*1}$. We didn't set the `return_sequences` option for this layer and by default in Keras it is `False`. Therefore, only the last cell will sends its results as the output of this layer. This output is a representation of out input data and we call it embedded features. It is a vector of shape (1,7). This is the output of encoder part of our model.

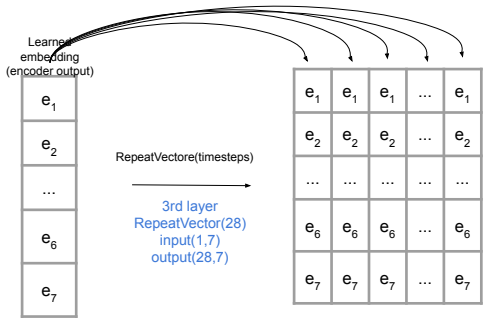


Figure 3.10: Autoencoder: connecting encoder to decoder using `RepeatVector()` layer

To be able to feed the encoder output to our decoder model, we need to transform it to a 3D shape. This is done by duplicating this vector using `RepeatVector(timesteps)`. Figure 3.10 illustrates this step. The output

of this step will have the shape of (1, 28, 7). This layer doesn't have any weight matrices.

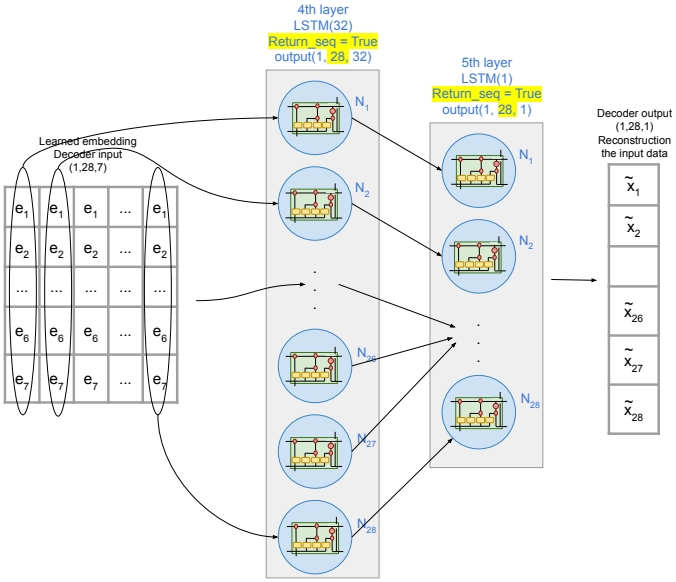


Figure 3.11: Autoencoder: Decoder model

Now the embedded features are ready to be fed to the decoder part of the model. Figure 3.11 is illustrating the decoder part of the model. As mentioned, the input to the decoder has the shape of (1, 28, 7) which means the fourth layer, `LSTM(32)`, should be unfolded 28 times. The size of the matrices for this layer will be as follow:

$$W_{xh} \in \mathbb{R}^{32*7},\ W_{hh} \in \mathbb{R}^{32*32},\ b \in \mathbb{R}^{32*1}$$

and the final size for all the four LSTM gates will be: $W_{xh} \in \mathbb{R}^{128*7}$, $W_{hh} \in \mathbb{R}^{128*32}$, $b \in \mathbb{R}^{128*1}$. We set the `return_sequences` to be `True` which makes the output to have a 3D shape of (1, 28, 32). The fifth and the last layer is `LSTM(1)` with `return_sequences` option set to `True`. The output of this layer will be the output of our autoencoder that tries to reconstruct the input data. Weight matrices for this layer are as follow:

$$W_{xh} \in \mathbb{R}^{1*32},\ W_{hh} \in \mathbb{R}^{1*1},\ b \in \mathbb{R}^{1*1}$$

and the final size for all the four LSTM gates will be: $W_{xh} \in \mathbb{R}^{4*32}$, $W_{hh} \in \mathbb{R}^{4*1}$, $b \in \mathbb{R}^{4*1}$.

3.2.3.2 Prediction model

After training the reconstruction Autoencoder, the output of encoder part of the model can be used as input to a prediction model. The proposed model in [9] and [1] is a combination of these two models. First, a two layer LSTM Autoencoder (encoder part of the model) for automatic feature extraction and representation learning (encoded features), as described in previous section. Then this embedded features will be fed to an MLP prediction model. The goal of using Autoencoder is to learn the generic behaviour across multiple time series with variety of patterns. It will capture the correlation among them and extracts the useful features in an encoded representation with a fixed dimension. More importantly, if there is an abnormal behaviour in the input, it will be captured by encoder and will be fed to prediction part of the model through learned embedded features. The authors in [1] mentioned that after training the Autoencoder, they used the last LSTM `cell_state` of the encoder and fed to their prediction model to do the actual prediction based on what Autoencoder has learned from past.

Due to implementation complexity of this approach, and since [1] doesn't explain some important details of their approach, we tried a different way and used the encoder output instead. We first fed our data to encoder and got the encoded representation of data using `encoder_model.predict()`. Then we used this encoded data as input to an LSTM prediction network (instead of MLP prediction model). Figure 3.12 shows the combination of the autoencoder and prediction model. The design of this model was inspired from [72].

The Keras code for this model can be found in listing 3.4. The input data first goes through the trained LSTM layers of encoder model. This will generate a new representation of the input data. This new encoded representation of the data will be fed to an LSTM model as the next step to make the final prediction. Notice that we are setting the `training` option to `True` which will be explained in the next section.

Listing 3.4: Prediction + Autoencoder Model in Keras

```
timesteps = 28
n_features = 1
dropout = 0.1
recurrent_dropout = 0.1
output_dropout = 0.1
h1 = 28
h2 = 14
h3 = 7

encoder_model = load_model(pretrained_encoder_model)
encoded_data, hidden_state, cell_state = encoder_model.predict(data)
```

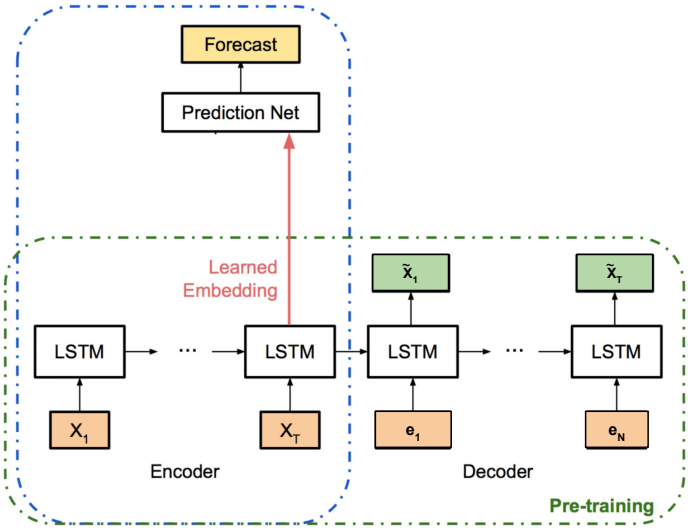


Figure 3.12: A prediction model based on feature extracted (e_N) by an autoencoder model, Picture adopted from [1]

```
input_layer = Input(batch_shape=(batch_size
                          , encoded_data.shape[1]
                          , encoded_data.shape[2]))
first_lstm = LSTM(h1, return_sequences=True
                  , dropout=dropout
                  , recurrent_dropout=recurrent_dropout
                  , activation='tanh')(input_layer, training=True)
if output_dropout:
    first_lstm = Dropout(output_dropout)(first_lstm, training=True)
second_lstm = LSTM(h2, return_sequences=False
                  , dropout=dropout
                  , recurrent_dropout=recurrent_dropout
                  , activation='tanh')(first_lstm, training=True)
if output_dropout:
    second_lstm = Dropout(output_dropout)(second_lstm, training=True)
first_dense = Dense(h3)(second_lstm)
out_layer = Dense(n_features)(first_dense)

ae_lstm_predict_model = Model(input_layer, out_layer)
```

Figure 3.13 illustrates the architecture of this model and how one instance of input data with shape of (1, 28, 1) would go through the network.

3.3 Prediction Uncertainty

To develop a robust uncertainty boundaries, authors of [1] combined Bootstrap and Bayesian approaches. In their work, they break down the prediction

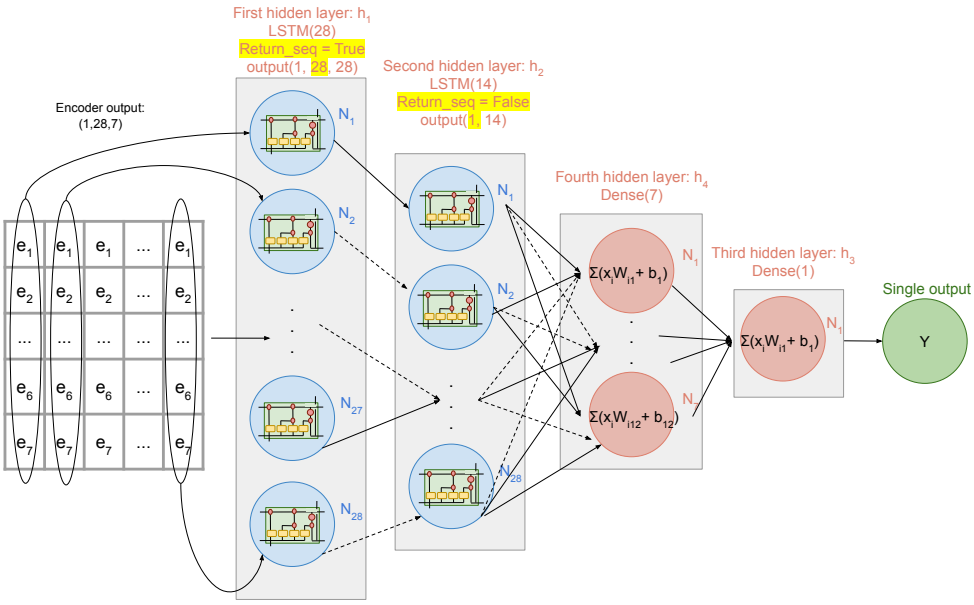


Figure 3.13: Our prediction model based on encoded representation of data

uncertainty into three different sources: Model uncertainty, Model misspecification and Inherent noise. They define model uncertainty as ignorance of the model parameters that can be reduced by adding more data. To capture this uncertainty, as it was mentioned before, our model is using a Bayesian neural network with Monte Carlo dropout (MC dropout) as variational inference approximation on posterior. This is done by adding dropout to all layers of encoder model and prediction model in the implementation. Therefore, the random dropout in encoder output, which is the prediction model's input, propagates to the prediction model.

We will use MC dropout at test time to apply sampling form posterior. As described in subsection 2.5.6, Keras has a `training` option that makes this possible for us. To implement this, we use layers from our trained prediction model and add `Dropout()` layers with a fixed dropout rate of 0.1. This will add a random dropout to each layer with probability of 0.1. We feed a test dataset to this new model and we get the output with stochastic dropouts at each layer. This process will be repeated B 10 times and the model output of each step can be seen as a random sample. At the end, we return the mean and variance of all these results as our model uncertainty.

A common challenge in time series anomaly detection is to differentiate abnormal trends from a normal but unusual trends, in a test sample which is different from training samples. To address this issue, authors of [1] sug-

gested the use of an autoencoder and they call the uncertainty that captures this "model misspecification". As was mentioned before, an autoencoder will learn important features of all training time series. This learned embedded layer will be the input for prediction model to do inference phase. Inside the LSTM layers of the encoder, variational dropout will be applied.

Algorithm 1, adopted from the paper from Uber, shows MC dropout algorithm that uses LSTM variational dropout for encoder and regular dropout for prediction model. It connects the two model and misspecification uncertainties to calculate the variance. The same algorithm in [1] also considers external features (e.g. wind speed, weather temperature, etc) that affect number of rides in a day and concatenate them into learned embedding. But we don't have such features for our project here. Our implementation code is also presented in listing 3.5

Algorithm 1: MC dropout: model and misspecification uncertainty

input : data x^*, encoder g(·), prediction network h(·), dropout probability p, number of iterations B

output prediction $\hat{y}^*_{mc}$, uncertainty η_1

:

for *b = 1 to B do* **do**

  $e^*_{(b)} \longleftarrow variationalDropout(g(x^*), p)$

  $\hat{y}^*_{(b)} \longleftarrow Dropout(h(e^*_{(b)}), p)$

end

// prediction

$\hat{y}^*_{mc} \longleftarrow \frac{1}{B}\sum_{b=1}^{B} \hat{y}^*_{(b)}$

// model uncertainty and misspecification

$\eta_1^2 \longleftarrow \frac{1}{B}\sum_{b=1}^{B} (\hat{y}^*_{(b)} - \hat{y}^*)^2$

return $\hat{y}^*_{mc}$, η_1

Listing 3.5: MC dropout implementation in Keras

```
#  Here model is the trained LSTM prediction model from previous section
    that gets encoder model's output as its input

timesteps = 28
n_features = 1
dr = 0.1
B = 10
y_hat_b = pd.DataFrame()

for b in range(0, B):
```

```
    mc_input = Input(batch_shape=(batch_size
                  , encoded_data.shape[1]
                  , encoded_data.shape[2]))
    first_lstm_layer = model.layers[1](mc_input)
    first_dropout = Dropout(dr)(first_lstm_layer, training=True)
    snd_lstm_layer = model.layers[3](first_dropout)
    snd_dropout = Dropout(dr)(snd_lstm_layer, training=True)
    third_dense_layer = model.layers[5](snd_dropout)
    output = model.layers[6](third_dense_layer)
    mc_model = Model(mc_input, output)
    prediction_result = mc_model.predict(test_data) # y_hat_star_b
    y_hat_b = y_hat_b.append(prediction_result)

y_hat_star_mc = y_hat_b.mean(level=[0, 1])
eta_1 = y_hat_b.var(level=[0, 1])
```

To estimate the third part of prediction uncertainty, inherent noise level in data, we get the residual sum of squares on a sample of test data as [1] suggests. Inherent noise is the inevitable natural noise in the data which depends on how the data was generated.

Algorithm 2 from Uber paper shows how the inference process is done. The final step is to combine calculated inherent noise and the MC dropout estimation. The challenge here was to decide how to combine these two estimations. We decided to consider the noise error from data as a constant value and add that to MC dropout estimation (model uncertainty + model misspecification). We will use this prediction interval to detect anomalies: whenever the predicted value falls outside the estimated interval, it will be considered as an anomaly. The code is listed in 3.6

Listing 3.6: Inherent noise estimation in Keras

```
y_hat_v = pd.DataFrame()
n_validation_sets = 10

for v in range(0, n_validation_sets):
    sampled_data = random_sample(test_data)
    prediction_result = model.predict(sampled_data) # y_hat_prime
    y_hat_v = y_hat_v.append(prediction_result)
eta_2 = y_hat_v.var()
```

Algorithm 2: Inference [1]

input : data x^*, encoder g(·), prediction network h(·), dropout probability p, number of iterations B

output prediction $\hat{y}^*$, predictive uncertainty uncertainty η

:

// prediction, model uncertainty and misspecificatio

$\hat{y}^*, \eta_1 \longleftarrow MCdropout(x^*, g, h, p, B)$

// Inherent noise

for x'_v *in validation set* $\{x'_1, ..., x'_V\}$ **do**

  $\hat{y}_v{}' \longleftarrow h(g(x'_v))$

end

$\eta_2^2 \longleftarrow \frac{1}{V} \sum_{v=1}^{V} (\hat{y}'_v - y'_v)^2$

// total prediction uncertainty

$\eta \longleftarrow \sqrt{\eta_1^2 + \eta_2^2}$

return $\hat{y}^*, \eta$

Chapter 4

Experiments and Results

4.1 Results

In this chapter, the results for our proposed model is presented. To evaluate the results, two baseline models are introduced: an MLP model and a vanilla LSTM model. We first go through the baselines results and then the results of the proposed model will be discussed. It should be mentioned that all the experiments and models were trained on HOPS cluster [74] from Logical Clocks [75]. The version of keras used to implement these models is 2.2.4-tf from TensorFlow API.

4.1.1 Baseline model #1: MLP model

For the MLP model with one-feature experiments, a combination of the following values listed in table 4.1 were tried:

Learning Rate	0.001, 0.01, 0.1, 0.5
Dropout Rate	0, 0.1, 0.14, 0.27, 0.36, 0.41, 0.5
Weight Decay	0, 0.1, 0.01

Table 4.1: Combination of different hyper-parameter values used for training the MLP model

Table 4.2 shows the settings used for the MLP model experiment. The model is trained on the training data and validated on the validation data set as described in section section 3.1. Table 4.3 shows the results for the best hyper-parameters selected on test data set. As this table shows, the best results for MLP with one feature belongs to the model that is using RMSprop optimizer,

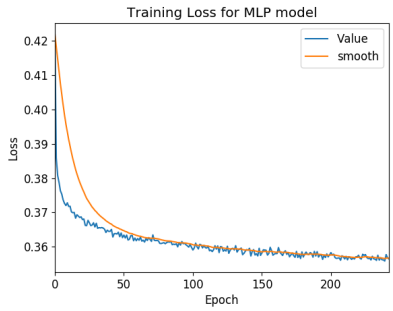
tanh as activation function, learning rate = 0.01, dropout rate = 0.1, and weight decay = 0.1. Figures 4.1a and 4.1b show the loss function during training the MLP model. We used `EarlyStopping` on validation with `patience=50` and `ModelCheckpoint` to save the best result.

Parameter	**Value**
Batch_size	138
Epochs	1000
Number of features	1
Train data shape	(49956, 28)
Validation data shape	(16422, 28)
Test data shape	(8004, 28)
First layer size	24, 30
Second layer size	12, 15
Output layer	1

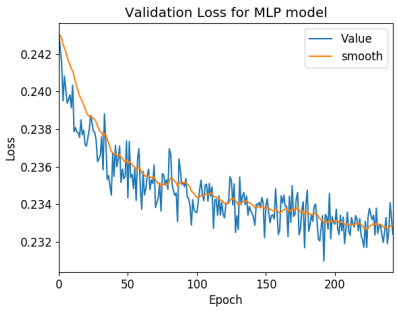
Table 4.2: Parameter settings for the MLP model

Optimizer	**Activation**	**BEST combination**	**Error on test (MSE)**
RMSprop	Relu	lr=0.5.dr=0.36.eta=0.01	0.248
Adam	Relu	lr=0.001.dr=0.1.eta=0	0.249
RMSprop	Tanh	lr=0.01.dr=0.1.eta=0.1	0.246
Adam	Tanh	lr=0.001.dr=0.1.eta=0.01	0.256

Table 4.3: Error on test data set after hyperparameter tuning for MLP models.



(a) Training Loss values



(b) Validation Loss values

Figure 4.1: Train and Validation loss function plots for the best results of table 4.3 for MLP model with one feature.

In chapter 3, we picked three time series as examples to follow up and illustrated them in figure 3.5. Figure 4.2 shows the prediction results of MLP model for these time series. The data in figure 4.2 corresponds to the green part (test data) of figure 3.5. The error rate for these predictions can be found in table 4.4.

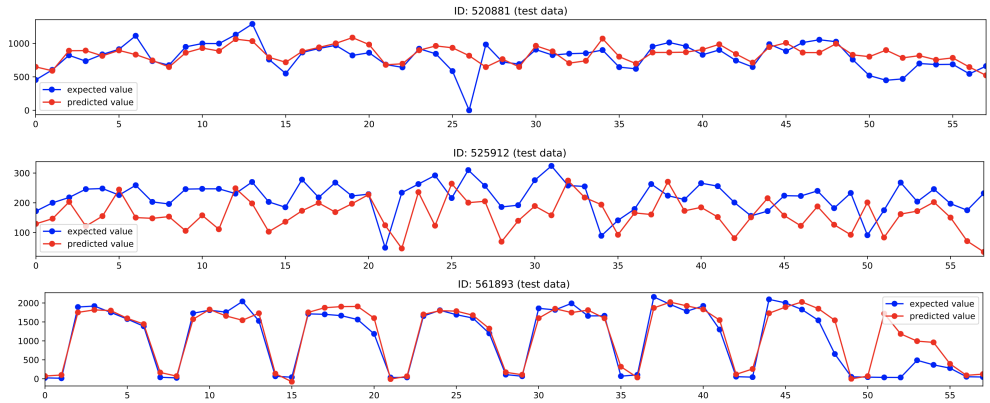


Figure 4.2: Prediction results of MLP one feature model for three time series samples of data. x-axis shows time steps in the test data and y-axis shows number of successful calls.

	MSE	RMSE
Train Error	0.358	0.598
Test Error	0.246	0.496

Table 4.4: The error rate on prediction results on train and test data sets.

4.1.2 Baseline model #2: vanilla LSTM for prediction

As our second baseline, the vanilla LSTM model from subsection 3.2.2 was trained using the combination of the following parameters in table 4.5:

Table 4.6 shows the error rate results for the best hyperparameters selected on the test data set. As it shows, the best results for vanilla LSTM model belongs to the model with size 50 neurons in the first LSTM layer and 25 neurons in second layer. It is using Adam optimizer, tanh as activation function, learning rate = 0.01 , dropout rate = 0.1 , and weight decay = 0.01. We also tried to train the model with RMSProp as optimizer and relu activation functions, but the model wasn't able to converge for most of the hyper-parameters. Figure 4.3a and 4.3b show the loss function for training and validation for 500 epochs.

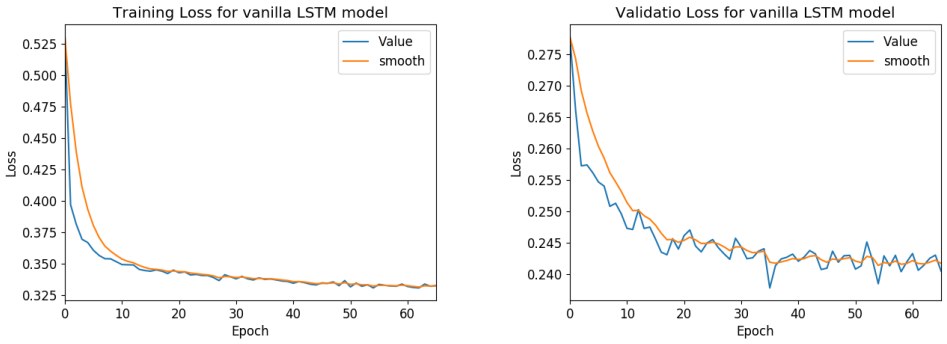
Learning Rate	0.001, 0.01, 0.1, 0.5
Dropout Rate	0, 0.1, 0.14, 0.27, 0.36, 0.41, 0.5
Weight Decay	0, 0.1, 0.01
First LSTM layer	28, 30, 50, 128
Second LSTM layer	7, 10, 15, 20, 25

Table 4.5: Combination of different hyper-parameter values and hidden network sizes that were used to train the vanilla LSTM model

We used `EarlyStopping` callback on validation data with `patience=30` and `ModelCheckpoint` to save the best weights.

Best hyper-parameters combination	lr=0.01.dr=0.1.eta=0.01
First LSTM Layer Size	50
Encoding Layer Size	25
Train Error	MSE=0.332 RMSE=0.576
Test Error	MSE=0.288 RMSE=0.537

Table 4.6: Hyperparameter tuning for vanilla LSTM prediction model and its results. It is using `tanh` activation function and `Adam` optimizer. The prediction errors for test and train datasets are also included.



(a) Training Loss values

(b) Validation Loss values

Figure 4.3: Vanilla LSTM Loss plots

Figure 4.4 shows the prediction results of the second baseline model for

our three example time series. The data in this figure corresponds to the green part (test data) of figure 3.5.

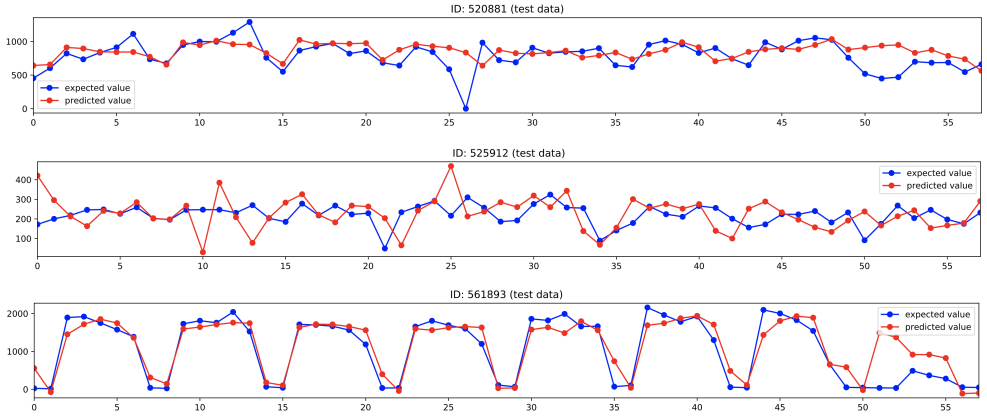


Figure 4.4: Prediction results of vanilla LSTM model for three time series samples of data. The x-axis is the number of timesteps in the test data

4.1.3 Our proposed AE + prediction model

Figure 4.5 summarizes the steps for our proposed model. As it shows, after preparing the data, we first train an autoencoder to reconstruct the input. Then we use the encoder part to extract the important features of the input data. We feed this encoded representation of data to a prediction model to get the number of calls for day 29, based on the history of the last 28 days. Finally, we estimate the uncertainty of the model using the novel solution from the Uber paper [1] and find the anomalies in time series that fall outside the 95% of the confidence interval. The following of this section describes the results of this method and in section section 4.2 we will discuss the uncertainty estimation results.

Figure 4.5: Summary of the proposed model

4.1.3.1 AE reconstruction

In this section we explain the work and results for training an autoencoder that reconstructs its input data. There were many details and options in building

the model that we will summurize here. As mentioned before, there are three different dropout rates that can be applied on a LSTM layer. Two of them are keras LSTM options: `recurrent_dropout` and `dropout`. The third one is applying a `Dropout()` layer after `LSTM()` layer which we call "output_dropout". During the hyperparameters tuning process, we are assigning the same value to all these three type of dropout options.

Before going to the details of the results, it should be mentioned that for autoencoder model, different architectures where tried as listed below:

- An autoencoder model with all of these three dropout options
- An autoencoder model with no `Dropout()` layer (without "output_dropout"). Only used keras LSTM dropout options: `recurrent_dropout` and `dropout`
- An autoencoder model with stateful LSTM layers, `stateful=True`
- An autoencoder model with stateless LSTM layers, `stateful=False`
- Different scenarios for the encoder model output, and how to connect it to decoder were also tested:
  - encoder output returns sequences, `return_sequences=True`
  - encoder output layer doesn't return sequences and the layers connected using `RepeatVector()`
- Optimizing the autoencoder model using adam optimizer
- Optimizing the autoencoder model using RMSprop optimizer
- Optimizing the autoencoder model using SGD optimizer (only for few experiments not all)
- Using `Relu` activation function
- Using `tanh` activation function

For the sake of brevity, only the model with the best results will be discussed here. The best results are from an autoencoder model with all three types of dropouts, stateful LSTM layers, and using `RepeatVector()` to connect encoder and decoder models. It is using `tanh` activation function and `RMSProp` as optimizer.

Similar to previous experiments, a combination of different hyper parameter values shown in table 4.7 are used to train the autoencoder model.

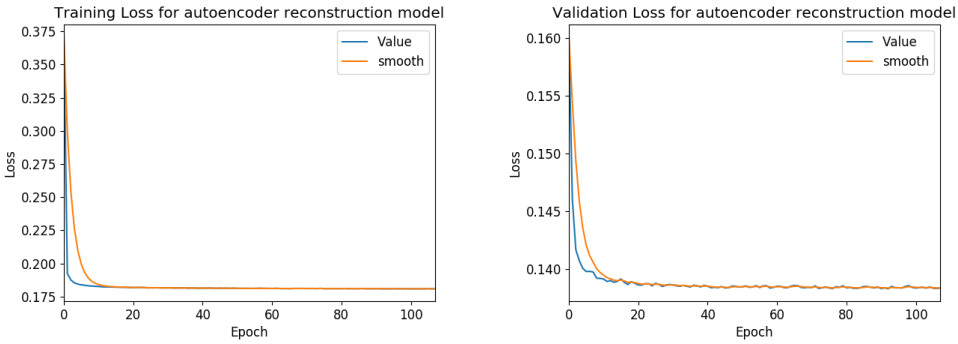
Learning Rate	0.001, 0.01, 0.1
Dropout Rate	0, 0.1, 0.14, 0.27, 0.36, 0.41, 0.5
Weight Decay	0, 0.1, 0.01
First LSTM layer	32, 50, 128
Encoding layer	7, 14, 20, 25, 28

Table 4.7: Combination of different hyper-parameter values and number of hidden units that were used to train the autoencoder model

After training these different models on training data sets and validating them using the validation data set, the model with least validation Loss was chosen as our the best autoencoder model. Table 4.8 shows the best model specifications. Figure 4.6a and 4.6b show the plots for Loss values for training and validation. The model has 128 hidden units for the first LSTM layer and 14 hidden units for the second LSTM layer (or encoding layer). The Loss metric shows the reconstruction error of applying the model on traing and test data set after 500 epochs. We used `EarlyStopping` callback on validation data with `patience=30`

Best hyper-parameters combination	lr=0.01, dr=0.1, eta=0.001
First LSTM Layer Size	128
Encoding Layer Size	14
Train Error	MSE=0.18074 RMSE=0.42514
Test Error	MSE=0.10502 RMSE=0.32407

Table 4.8: Hyperparameter tuning for AE reconstruction model. The dropout rate is the same for all three types of dropout: `recurrent_dropout`, `dropout`, and `Dropout` layer. It is using `tanh` activation function and `RMSProp` as optimizer.



(a) Training Loss values

(b) Validation Loss values

Figure 4.6: Reconstruction AE Loss plots

The results of reconstruction time series with autoencoder are illustrated in figure 4.7 for three examples. From 28 timesteps for each time series, these plots are only showing the last time step reconstruction for the three time series shown in section section 3.1. This is the results on test dataset. The whole timesteps reconstruction is shown in appendix A for only one of the time series with id=520881, in our test dataset.

This trained autoencoder is used for the next part of this experiment, to train a prediction model. The enoder model of our autoencoder is used to generate a new representation of our data set. The prediction model in the next section recieves that as its input and make predictions.

4.1.3.2 Prediction model

The trained autoencoder in the previous section consists of two parts: an encoder and a decoder. For prediction, we use the encoder part of the model that gives us an encoded representation of the input:

```
encoded_data = encoder_model.predict(data)
```

Table 4.9 lists the hyper parameter values that were used to train the prediction model.

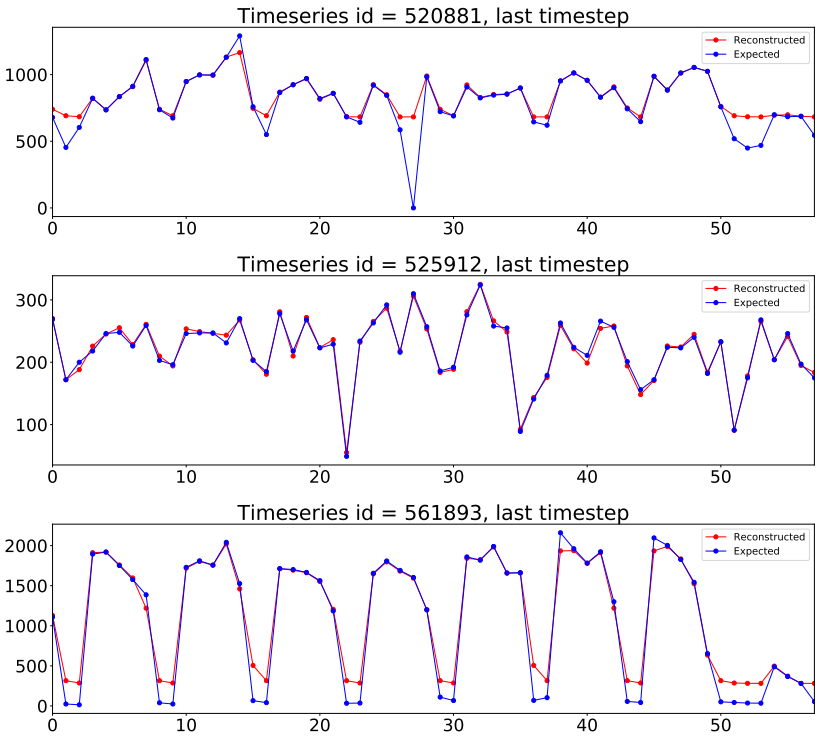


Figure 4.7: AE reconstruction of test data. These are the reconstruction of the last time step (28th) of three sample time series

Learning Rate	0.001, 0.01
Dropout Rate	0, 0.1, 0.14, 0.27, 0.36
Weight Decay	0, 0.001, 0.01, 0.1
First layer (LSTM)	128, 28
Second layer (LSTM)	64, 14
Third layer (Dense)	50, 7

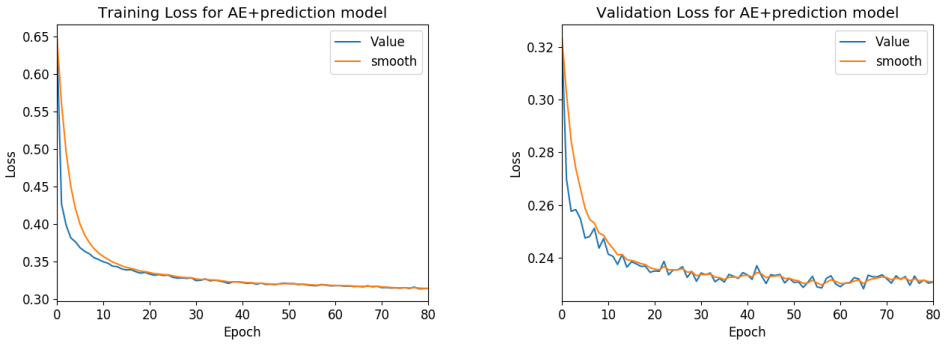
Table 4.9: Combination of different hyper-parameter values and number of hidden units that were used to train the prediction model based on AE encoded representation

The model was trained on the encoded presentation of the training data set and validated on the encoded representation of the validation data set. After training and tuning different models, the Loss value for encoded test data set for each model was evaluated. Table 4.10 shows the best combination of the hyper parameters for a prediction model with minimum Loss value on test data set.

Best hyper-parameters combination	lr=0.001, dr=0.1, eta=0
First LSTM Layer Size	128
Second LSTM Layer Size	14
First Dense Layer Size	7
Second Dense Layer Size	1
Train Error	MSE=0.25150 RMSE= 0.50150
Test Error	MSE=0.25049 RMSE=0.50049

Table 4.10: Hyper-parameter tuning for prediction model based on encoded data. The dropout rate is the same for all three types of dropout: `recurrent_dropout`, `dropout`, and `Dropout` layer. It is using `Relu` activation function and `RMSProp` as optimizer. The prediction error rates on test and train data are also included.

Figures 4.8a and 4.8b shows the loss function during training the prediction model based on encoded data. It was stopped at epoch 58 because of using `EarlyStopping` callback.



(a) Training Loss values

(b) Validation Loss values

Figure 4.8: Loss plots for prediction model based on encoded data

Figure 4.9 shows the results of this model predictions for three time series we saw before in test dataset. As mentioned before, the model looks at number of calls for the last 28 days and then makes a prediction for day 29. These plots are showing how the prediction for day 29 (red line) looks like in compare to the true value for number of calls on this day (blue line). Consider time series with id=520881, the top plot. For this time seris, the model predicted a value

between 500 to 1000 for timestep 27 (red line), but all of a sudden this value dropped to a very small number close to zero as the actual logs are showing (blue line). This should be detected as an anomaly since the data has a sudden change in its behavior. This will be explained in the next section in more details.

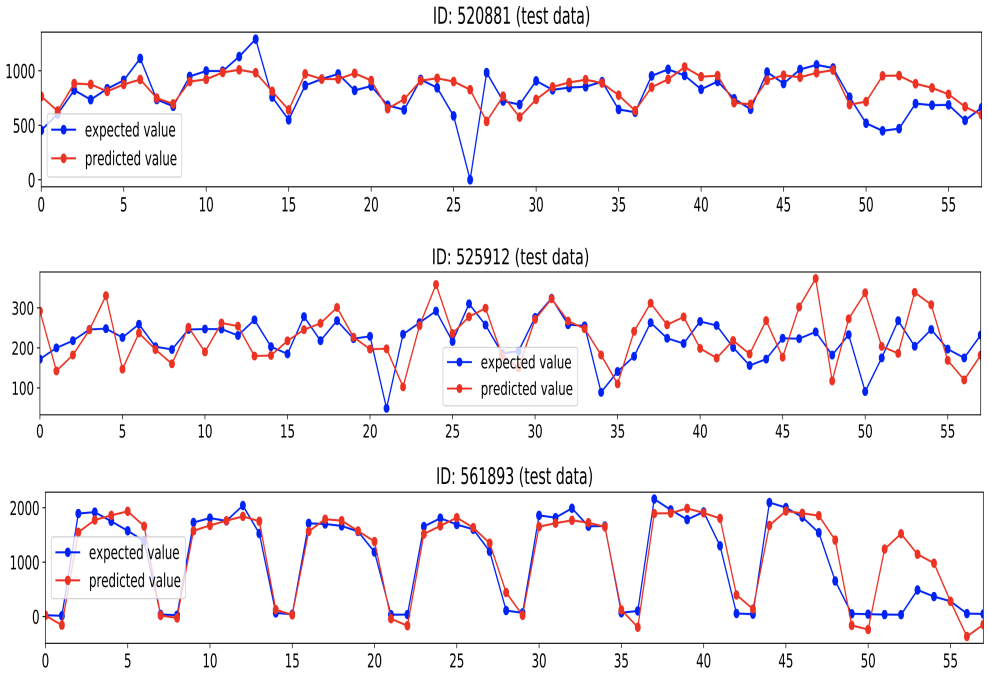


Figure 4.9: Prediction results for LSTM model based on encoded data. The plots show the predicted values for day 29 based on the history of last 28 days.

4.2 Uncertainty estimation

In section section 3.3 we described how the uncertainty is estimated for our proposed AE + prediction model.

Figures 4.10, 4.11, and 4.12 show the estimated predictive uncertainty in three time series from test data that we are following as example in this report. In these plots, there are B=10 light gray lines which are the prediction results from algorithm 1. The red line is the mean of these predictions. The gray interval is the estimated prediction interval meaning that whenever we predict the value for day 29 at each time step falls in these interval. When the new data comes in, which is presented by blue line, we use 95% of this interval and if it falls outside that area we would consider it as an abnormal data point and it

would raise an alarm. In other words, whenever the blue line deviate from the usual trend and behave in a different way, that deviation will be considered as an abnormal behaviour. This gray interval consists of two parts: `eta1` which is the second output of algorithm 1 is presenting model misspecification and model uncertainty (green dotted line in the plots), `eta2` which is the output of algorithm 2 is capturing the inherent noise (pink dotted line in the plots). Appendix B shows the same plots for uncertainty estimation without showing the boundaries for `eta1` and `eta2`.

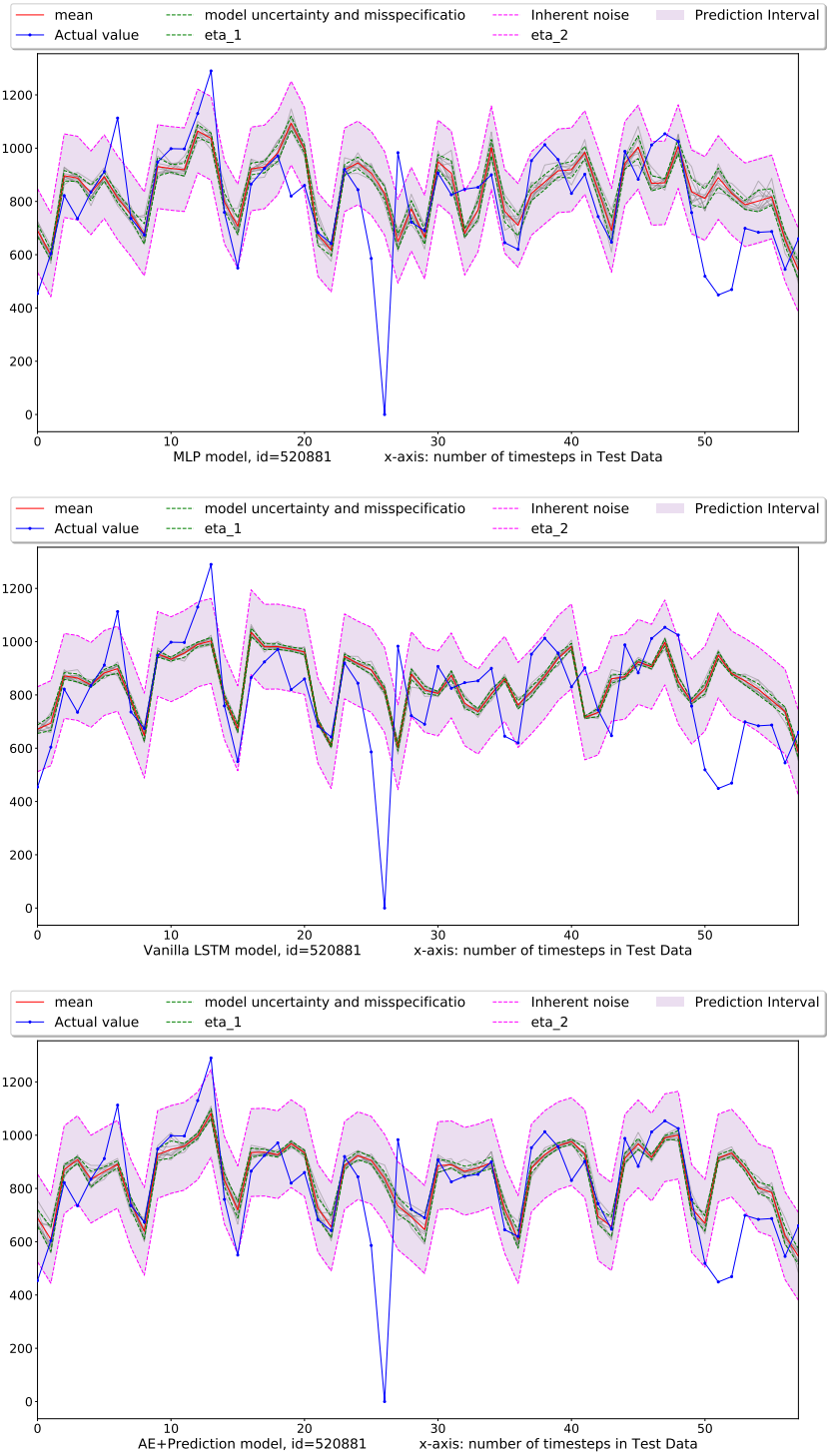


Figure 4.10: Estimated prediction interval by three models on test data for time series with ID=520881. The gray lines are the B=10 predictions from algorithm 1 and red line is the mean of all these B=10 predictions. The blue line shows the actual data points at each timestep. `eta1` is the error calculated in algorithm 1 and `eta2` is the inherent noise error based on algorithm 2.

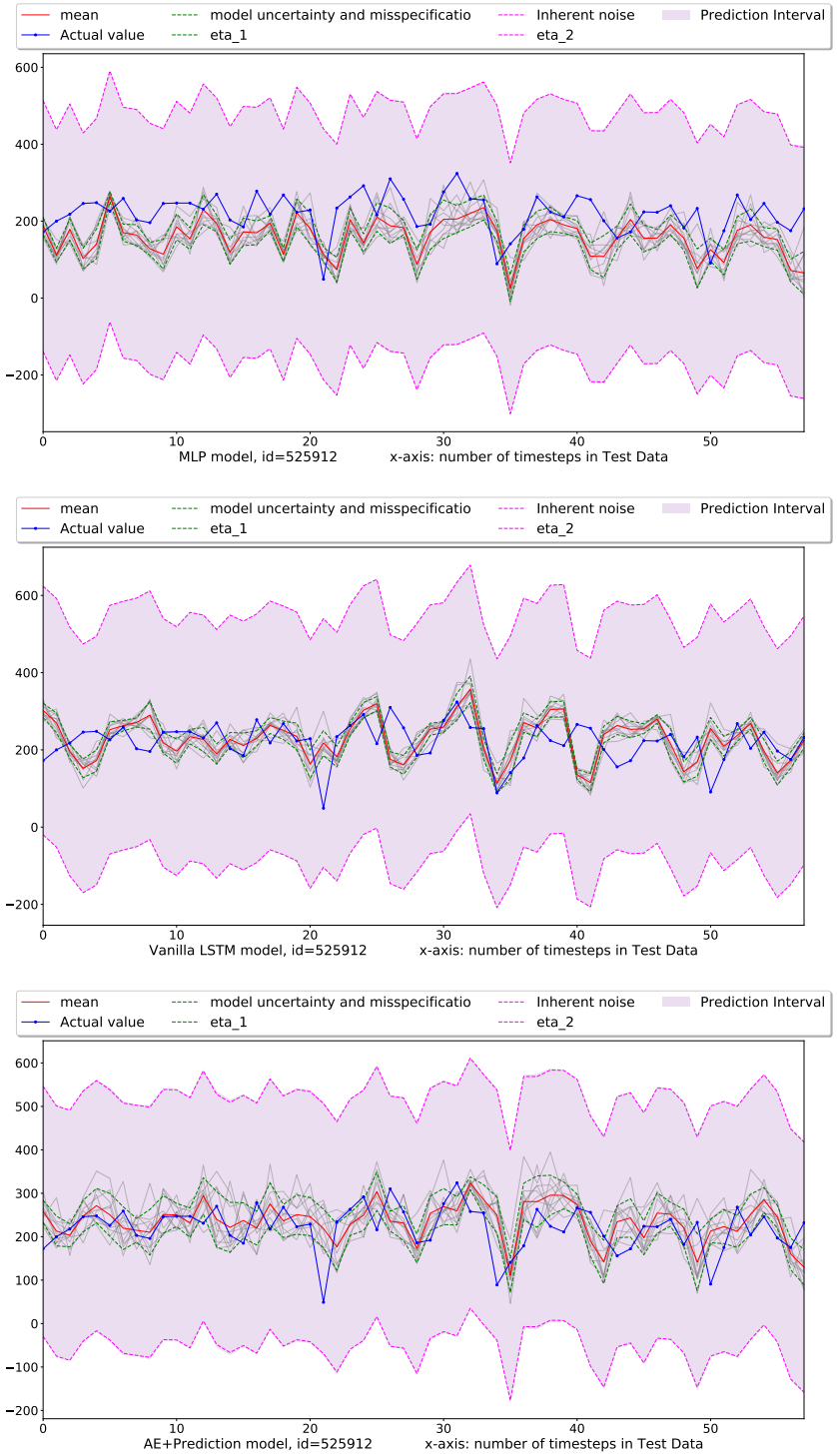


Figure 4.11: Estimated prediction interval by three models on test data for time series with ID=525912. The gray lines are the B=10 predictions from algorithm 1 and red line is the mean of all these B=10 predictions. The blue line shows the actual data points at each timestep. `eta1` is the error calculated in algorithm 1 and `eta2` is the inherent noise error based on algorithm 2.

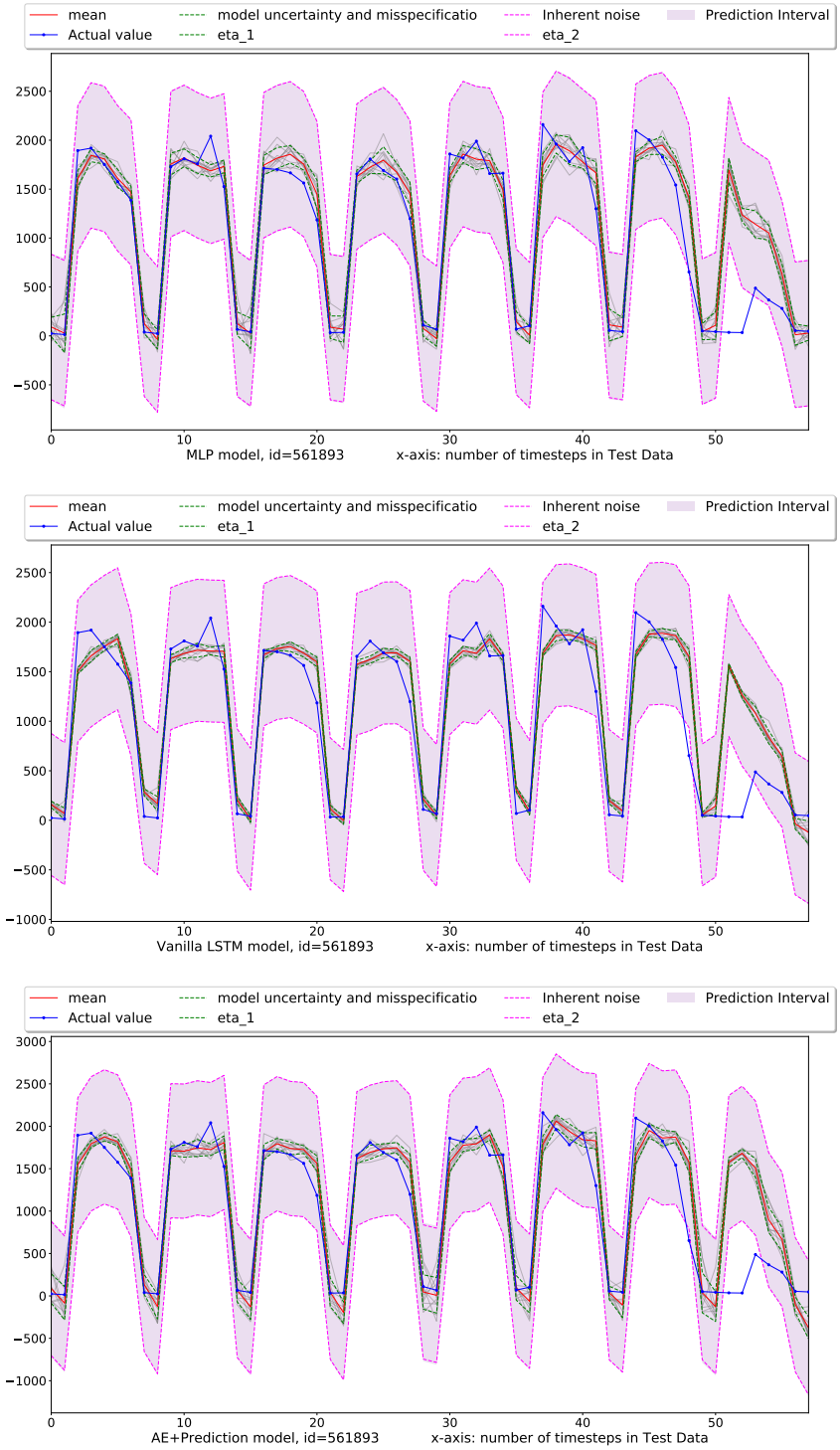


Figure 4.12: Estimated prediction interval by three models on test data for time series with ID=561893. The gray lines are the B=10 predictions from algorithm 1 and red line is the mean of all these B=10 predictions. The blue line shows the actual data points at each timestep. `eta1` is the error calculated in algorithm 1 and `eta2` is the inherent noise error based on algorithm 2.

4.3 Anomaly detection

One important use-case of the uncertainty estimation is to detect unusual patterns in the time series. We use the estimated predictive uncertainty to detect abnormal behaviour in our test data. The confidence interval gives us a boundary based on population mean and standard deviation. We use 95% of this boundary saying that the true values of our estimation is most likely to be in this range. We call any data point that falls outside this interval an "abnormal" data observation. To be able to evaluate the results of the detected anomalies, we labeled the test data with abnormal behaviours with the help of domain experts. These labels are shown in figures 4.13 and 4.14 with green dots and the detected anomalies are marked with red dots. For each time series, these figures show the anomalies detected by our three models: a) MLP model, b) Vanilla LSTM model, and 3) our proposed autoencoder based prediction model.

Figure 4.13 shows the detected anomalies for time series with ID 520881. For this specific time series, the MLP and vanilla LSTM models (top and middle plots) are having much more false positives in comparison to our proposed model. Figure 4.14 shows these results for time series with ID 561893. For this time series, all three models have almost the same results, only the proposed model has a higher true positive rate. For our third time series, with ID 595212, there is no anomalies detected which is correct. No abnormal data points were labeled for this time series. Looking at this time series plotted in picture 3.5 also clearly shows that there is no sudden changes in this time series behaviour.

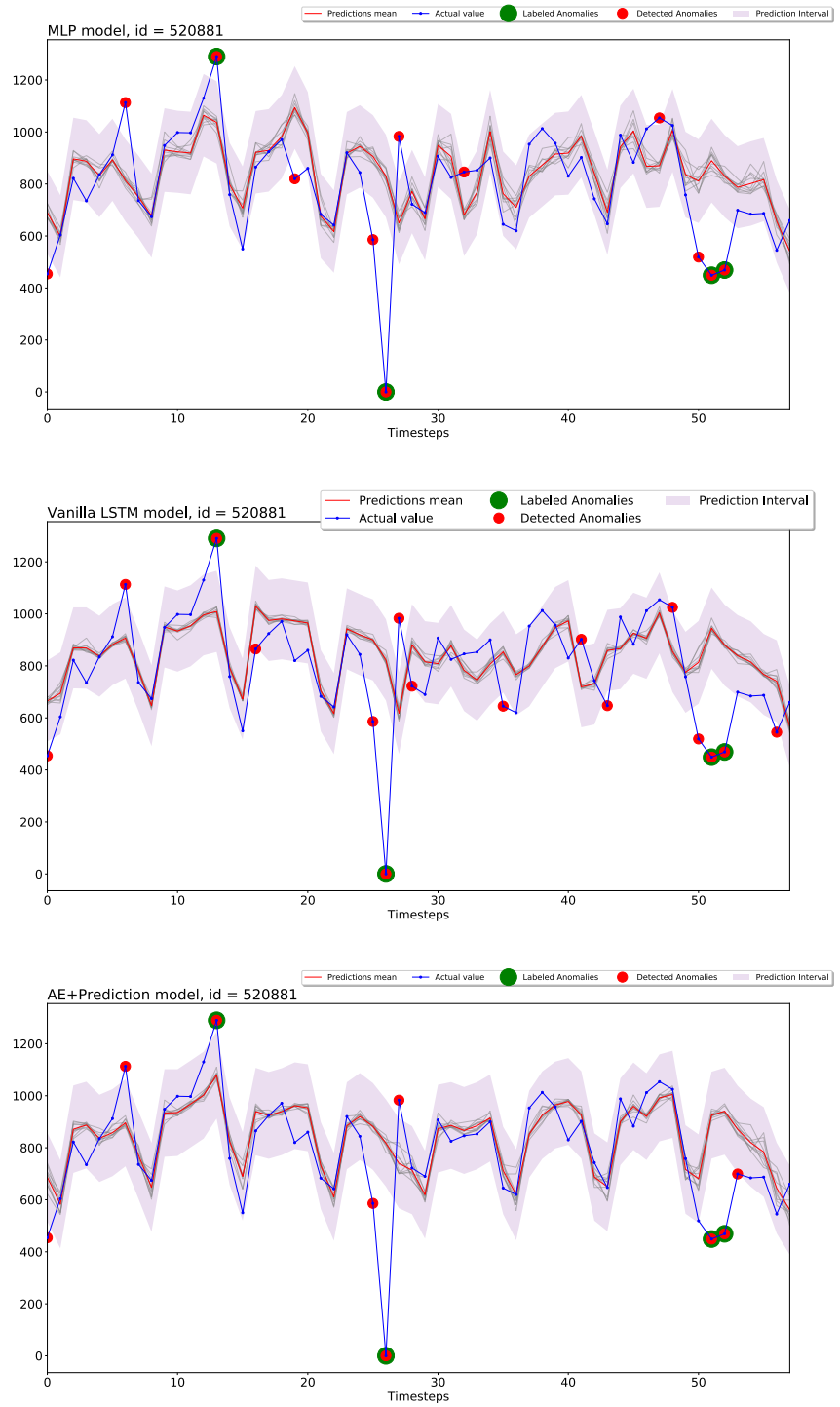


Figure 4.13: Detected anomalies (the red dots) for time series with ID=520881 by the two based line prediction models and our proposed model based on autoencoder extracted features

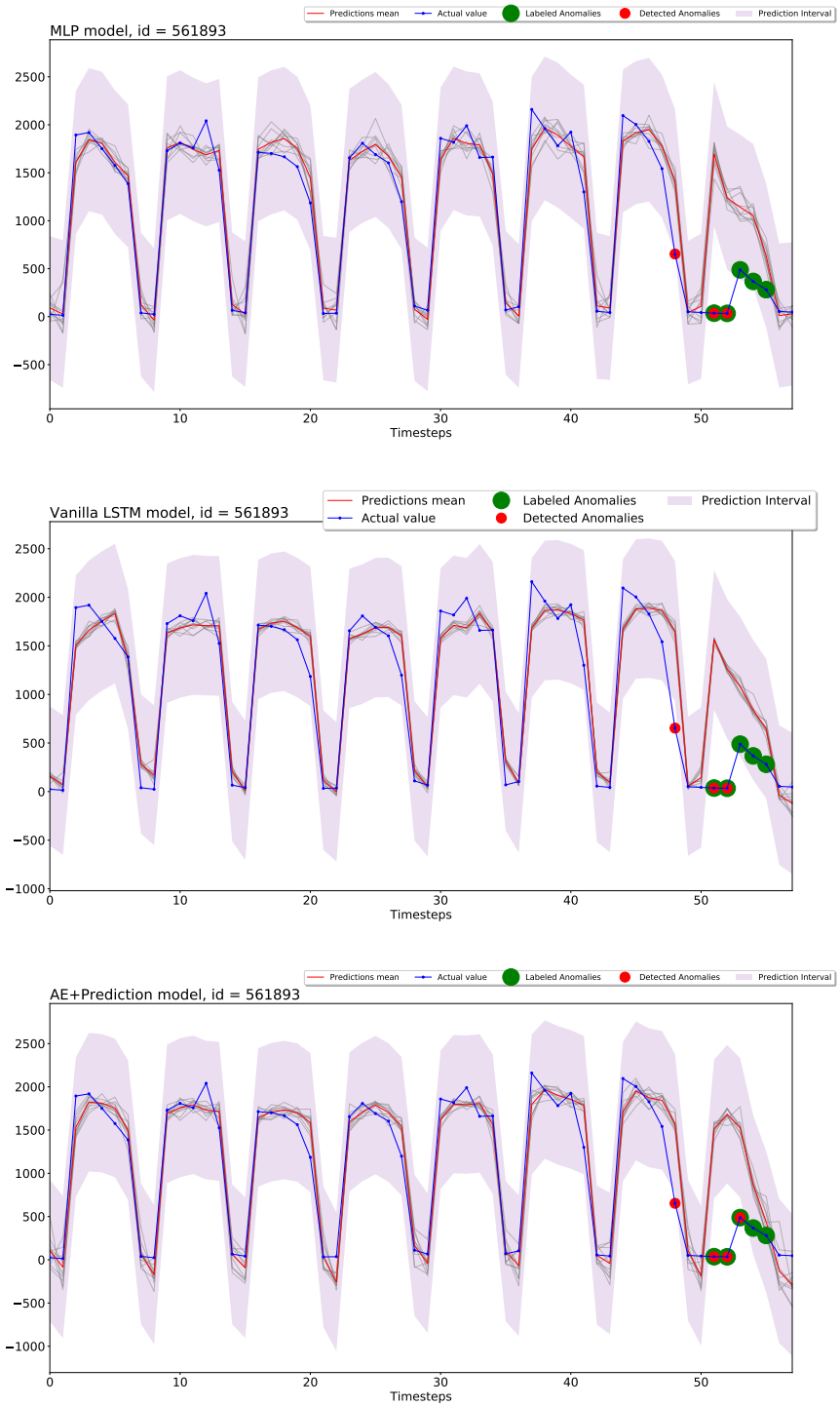


Figure 4.14: Detected anomalies (the red dots) for time series with ID=561893 by the two based line prediction models and our proposed model based on autoencoder extracted features

4.4 Discussion and results summary

Autoencoder reconstruction results

The results for reconstructing time series using autoencoder is illustrated in figure 4.7. As it shows, this model can successfully reconstruct the usual pattern of each time series. For time series with ID 525912, which doesn't have any sudden changes in its test data, it almost reconstruct the input perfectly. For ID 520881, the autoencoder model can reconstruct the usual pattern very well but it fails to detect the sudden changes in this time series. In general, the autoencoder can successfully reconstruct the usual pattern in almost all time series.

Prediction results

Table 4.11 is putting together the prediction error rates reported in 4.4, 4.6, and 4.10 tables. [1] As you can see, the prediction results for all three models are quite close and there is no big difference on their prediction error rates. Our proposed model is slightly better than the base lines. MLP model has surprisingly very close outcome to the LSTM models. This can be due to complexity of data: our data consists of simple time series and doesn't have long-range temporal dependencies. Therefore, a few recent timesteps are probably enough for predicting the next day's value. By comparing the prediction plots in figures 4.2 for MLP, 4.4 for vanilla LSTM, and 4.9 for AE+prediction models, we can see that all three models are able to follow the time series trends quite well and they all fail to predict the sudden changes in time series. Appendix C is putting the predictions of different models for each time series next to each other to make it easier to compare the results. Our AE+prediction model has a slightly better prediction performance.

[1] As mentioned in Chapter 3, these models are trained on data normalized using z-score method. Before that, we first tried to scale data using min-max scaler to be between 0 to 1. We could get very good prediction results for MLP model trained on this data. The error rate was (MSE=0.01130, RMSE=0.10630) on training data set, and (MSE=0.00759, RMSE=0.08710) on test data set. Unfortunately, the LSTM models were failing to train on min-max scaled data. Therefore, we used z-score normalization for training all the models.

		MSE	RMSE
MLP	Train	0.358	0.598
	Test	0.246	0.496
Vanilla LSTM	Train	0.332	0.576
	Test	0.288	0.537
AE+Prediction	Train	0.251	0.501
	Test	0.250	0.500

Table 4.11: Comparing the prediction error with our benchmarks. Our proposed model has the lowest error rate than the benchmarks. Vanilla LSTM model is slightly better than the MLP model but they are quite close.

In comparison to the results from Uber's paper [1], the authors mentioned they were feeding some external features (like weather condition, wind speed, etc that affect the number of daily rides) to their prediction model beside the learned embedding from enocder-decoder model. They did not explain what exactly these external features are. It is possible that apart from weather condition, they also helped the model by giving information about holidays and special events days. All these external features could help the prediction model to generate better results. As mentioned before, our time series are just simple information about number of daily calls and we don't have any other source of information. Apart from that, in [1], an encoder-decoder model that predicts the next day's value is used for extracting important features of the data. This way, the encoder model would capture the features that are more useful to have an accurate prediction. But in our proposed model, we used an autoencoder for reconstructing the input data which will cause the encoder to have a different approach for feature extraction.

Another interesting thing to notice in table 4.11 is that the test error is smaller than the train prediction error. It seems like our test data is simpler than the training data and it is easier to find the trends for the test data set. This could be due to the way we created our train, test, and validation data sets illustrated by figure 3.3. As this picture shows, our training data is from a completely different time of the year than the test (and validation) data. This will cause a temporal bias as the training data has a different distribution in compare to test data distribution. In other word, our training data is the data from June 2016 to June 2017, the validation data belongs to July to October 2017, and test data is almost the last two month of the year. This happened because the available data was small and we only had data for 138 time series, we couldn't afford to keep some time series as completely unseen data for test.

Also, we were relying on autoencoder to capture these temporal biases.

Anomaly detection

As mentioned before, we labeled the test data set with true abnormal data points. As the results show in figures 4.13 and 4.14, for each specific time series our baselines and proposed models have a different rate of false and true positives. For some time series, there is a higher rate of false positive and for some it is lower. In general, we observed that the MLP model has a higher false positive rate in compare to the two other models. Figure 4.15 shows the precision/recall plot for these three models and figure 4.16 shows their ROC curves. As these plots are showing, MLP model has the lowest performance and vanilla LSTM and AE+prediction model are very close in detecting abnormal data points. In the precision/recall plot, vanilla LSTM and AE+prediction have a better precision rate and their related lines are above the MLP line. Also, the Area Under Curve (AUC) for our proposed model is slightly better than the baseline models meaning that it has a higher rate of true positives in compare to the other two models, as shown in figure 4.16.

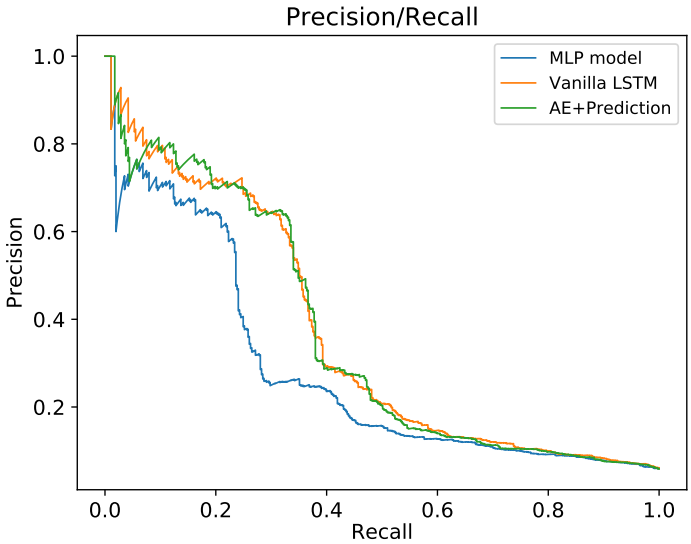


Figure 4.15: Precision/Recall for two baselines and the proposed model

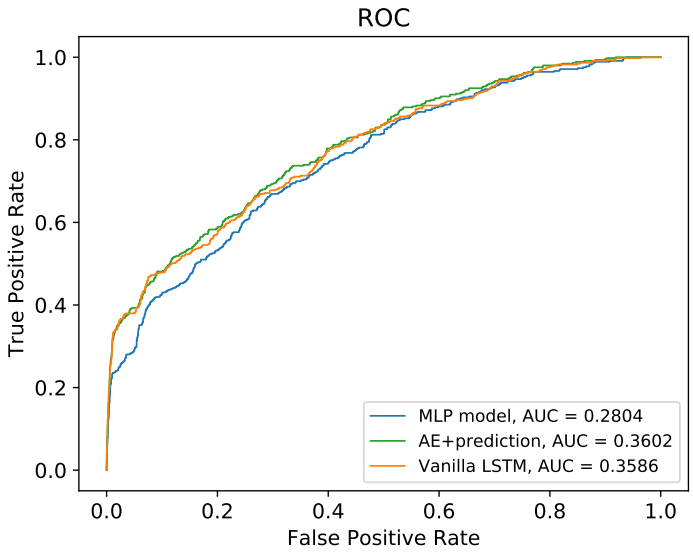


Figure 4.16: ROC curve for two baselines and the proposed model

4.5 Unseen data, model generalization

The Uber researchers in [9] claim that their model generalize very well on unseen data for time series prediction. Here we decided to apply the proposed model on data repository from numenta group, NAB [2]. Figure 4.17 illustrates one of the traffic time series from this data set called 'realTraffic\TravelTime_387.csv'. We took the data from NAB repository, scaled the data with z-score normalization and fed it to our encoder model. The result is encoded representation of this data which is given to our LSTM prediction model as its input.

[2]https://github.com/numenta/NAB/tree/master/data

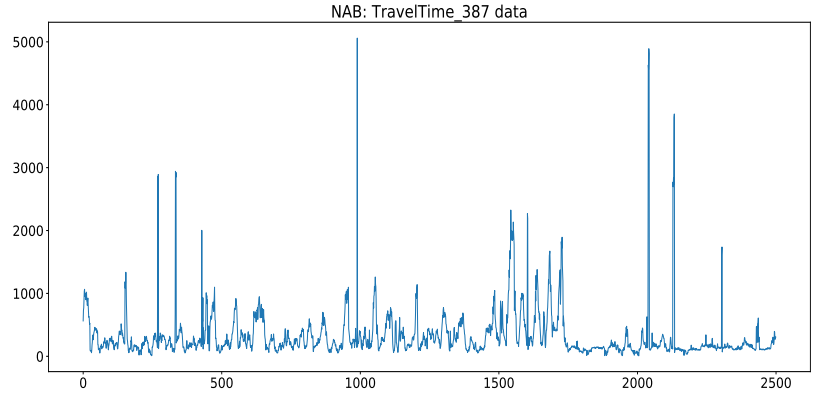


Figure 4.17: A traffic time series from NAB data repository: 'realTraffic\TravelTime_387.csv'

Figure 4.18 shows the prediction result and estimated uncertainty interval for part of this time series around 1000th timestep. The reason of plotting only part of the time series is just to have a clear and easy to understand picture. As the plot shows, our proposed model was able to follow the usual pattern of this time series quite well. Similar to our previous experiments, it was not able to predict the abnormal behaviour in this data. The estimated uncertainty interval however, helps to detect the unusual changes in time series' trend. This shows that our model can generalize very well and can be applied on unseen data as well.

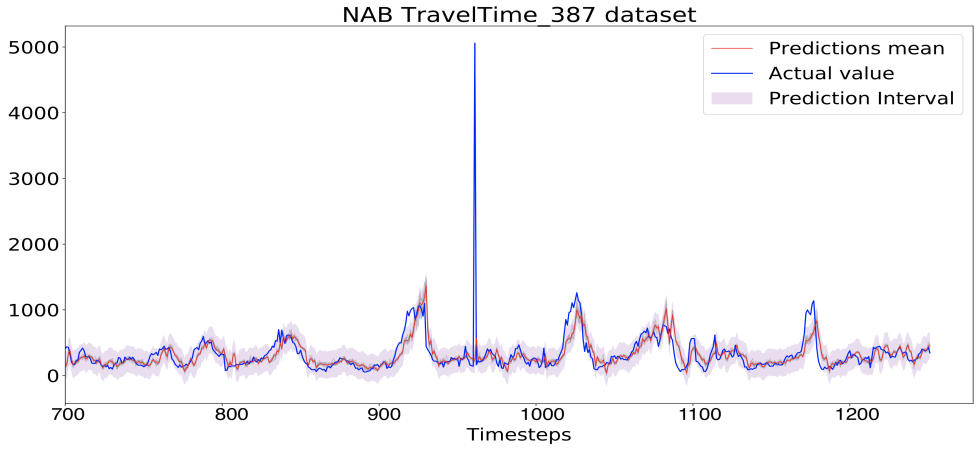


Figure 4.18: Estimated prediction interval on unseen data from NAB repository

Chapter 5

Future work

The proposed model in this thesis for anomaly detection could be still tested with some other configurations to see if we can get better results. One of the future studies could be repeating the experiment with different lookback window sizes and investigate the effect of it on prediction results. We used 28 days as lookback window, it is interesting to test smaller time window like 7 or 14 days as well. Another experiment is to apply this model on different data sets and compare the results.

To get the encoded representation of the data, we used an autoencoder that reconstructs its input in this work. One could train an autoencoder that predicts the next timesteps instead of reconstructing the input. This way, the encoder model would learn important features of the data to do a better prediction. It would be interesting to compare the results of prediction model based on these two different autoencoders.

Investigating the effect of adding MC dropout is also a valuable study to evaluate the quality of the uncertainty estimation method. A similar idea is used in [1] by comparing different scenarios as follow:

- Not using dropout layers in Encoder, but prediction model would have dropout layers and estimate only model uncertainty by applying MC dropout on prediction model
- Not considering inherent noise (η_1 in algorithm, 2), using MC dropout in both Encoder and prediction model

Bibliography

[1] Lingxue Zhu and Nikolay Laptev. "Deep and confident prediction for time series at uber". In: *Data Mining Workshops (ICDMW), 2017 IEEE International Conference on*. IEEE. 2017, pp. 103–110.

[2] *Ultimate guide to building a machine learning anomaly detection system*. Tech. rep. Anodot, 2017. URL: www.anodot.com.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey". In: *ACM computing surveys (CSUR)* 41.3 (2009), p. 15.

[4] Manish Gupta et al. "Outlier detection for temporal data: A survey". In: *IEEE Transactions on Knowledge and Data Engineering* 26.9 (2014), pp. 2250–2267.

[5] Guilherme O Campos et al. "On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study". In: *Data Mining and Knowledge Discovery* 30.4 (2016), pp. 891–927.

[6] Valentin Flunkert, David Salinas, and Jan Gasthaus. "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *arXiv preprint arXiv:1704.04110* (2017).

[7] Laurens De Haan and Ana Ferreira. *Extreme value theory: an introduction*. Springer Science & Business Media, 2007.

[8] *NIST/SEMATECH e-Handbook of Statistical Methods*. NIST. 2012. URL: https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc444.htm.

[9] Nikolay Laptev et al. "Time-series extreme event forecasting with neural networks at uber". In: *International Conference on Machine Learning*. 34. 2017, pp. 1–5.

[10] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[11] Tobias Lang and Matthias Rettenmeier. "Understanding consumer behavior with recurrent neural networks". In: *Workshop on Machine Learning Methods for Recommender Systems*. 2017.

[12] Chao Huang et al. "DeepCrime: attentive hierarchical recurrent networks for crime prediction". In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM. 2018, pp. 1423–1432.

[13] Jason Brownlee. *What is Deep Learning*. 2016. URL: https://machinelearningmast
com/what-is-deep-learning/.

[14] Francois Chollet. *Deep learning with python*. Manning Publications Co., 2017.

[15] Yoshua Bengio. *Foundations and Challenges of Deep Learning*. Deep Learning School. 2016. URL: https://www.youtube.com/watch?v=11rsu_WwZTc&t=1091s.

[16] Yoshua Bengio et al. "Greedy layer-wise training of deep networks". In: *Advances in neural information processing systems*. 2007, pp. 153–160.

[17] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[19] Andrew Zisserman. *Self-Supervised Learning*. 2018. URL: https://project.inria.fr/paiss/files/2018/07/zisserman-self-supervised.pdf.

[20] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[21] Diederik P Kingma and Max Welling. "Stochastic gradient VB and the variational auto-encoder". In: *Second International Conference on Learning Representations, ICLR*. 2014.

[22] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[23] Mark A Kramer. "Nonlinear principal component analysis using autoassociative neural networks". In: *AIChE journal* 37.2 (1991), pp. 233–243.

[24] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[25] Ali Yasoubi, Reza Hojabr, and Mehdi Modarressi. "Power-efficient accelerator design for neural networks using computation reuse". In: *IEEE Computer Architecture Letters* 16.1 (2017), pp. 72–75.

[26] Serena Yeung Fei-Fei Li Justin Johnson. *cs231n Lecture notes on Recurrent Neural Networks*. May 2017.

[27] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[28] Jason Brownlee. *Difference Between Return Sequences and Return States for LSTMs in Keras*. 2017. URL: https://machinelearningmastery.com/return-sequences-and-return-states-for-lstms-in-keras/.

[29] Akash Singh. *Anomaly detection for temporal data using long short-term memory (lstm)*. 2017.

[30] Yarin Gal. "Uncertainty in deep learning". PhD thesis. PhD thesis, University of Cambridge, 2016.

[31] Yarin Gal and Zoubin Ghahramani. "A theoretically grounded application of dropout in recurrent neural networks". In: *Advances in neural information processing systems*. 2016, pp. 1019–1027.

[32] Xu Yan et al. "Classifying relations via long short term memory networks along shortest dependency path". In: *arXiv preprint arXiv:1508.03720* (2015).

[33] Y-Lan Boureau, Sumit Chopra, Yann LeCun, et al. "A unified energy-based framework for unsupervised learning". In: *Artificial Intelligence and Statistics*. 2007, pp. 371–379.

[34] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[35] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[36] David Charte et al. "A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines". In: *Information Fusion* 44 (2018), pp. 78–96.

[37] *Autoencoder*. URL: https://en.wikipedia.org/wiki/Autoencoder.

[38] Mohammad Assaad, Romuald Boné, and Hubert Cardot. "A new boosting algorithm for improved time-series forecasting with recurrent neural networks". In: *Information Fusion* 9.1 (2008), pp. 41–55.

[39] Charles Blundell et al. "Weight uncertainty in neural networks". In: *arXiv preprint arXiv:1505.05424* (2015).

[40] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. "Variational inference: A review for statisticians". In: *Journal of the American statistical Association* 112.518 (2017), pp. 859–877.

[41] Christophe Andrieu et al. "An introduction to MCMC for machine learning". In: *Machine learning* 50.1-2 (2003), pp. 5–43.

[42] Adriano Azevedo-Filho and Ross D Shachter. "Laplace's method approximations for probabilistic inference in belief networks with continuous variables". In: *Uncertainty Proceedings 1994*. Elsevier, 1994, pp. 28–36.

[43] John Paisley, David Blei, and Michael Jordan. "Variational Bayesian inference with stochastic search". In: *arXiv preprint arXiv:1206.6430* (2012).

[44] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning". In: *international conference on machine learning*. 2016, pp. 1050–1059.

[45] Charu C Aggarwal. "Outlier analysis". In: *Data mining*. Springer. 2015, pp. 237–263.

[46] Victoria Hodge and Jim Austin. "A survey of outlier detection methodologies". In: *Artificial intelligence review* 22.2 (2004), pp. 85–126.

[47] Yang Zhang, Nirvana Meratnia, and Paul JM Havinga. "Outlier detection techniques for wireless sensor networks: A survey." In: *IEEE Communications Surveys and Tutorials* 12.2 (2010), pp. 159–170.

[48] Mingyan Teng. "Anomaly detection on time series". In: *2010 IEEE International Conference on Progress in Informatics and Computing*. Vol. 1. IEEE. 2010, pp. 603–608.

[49] JP Burman and MC Otto. "Census bureau research project: Outliers in time series". In: *Bureau of the Census, SRD Res. Rep. CENSUS/SRD/RR-88114* (1988).

[50] Anthony J Fox. "Outliers in time series". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 34.3 (1972), pp. 350–363.

[51] Raghavendra Chalapathy and Sanjay Chawla. "Deep learning for anomaly detection: A survey". In: *arXiv preprint arXiv:1901.03407* (2019).

[52] Rose Yu et al. "Long-term forecasting using tensor-train rnns". In: *arXiv preprint arXiv:1711.00073* (2017).

[53] George EP Box et al. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[54] Lawrence R Rabiner and Biing-Hwang Juang. "An introduction to hidden Markov models". In: *ieee assp magazine* 3.1 (1986), pp. 4–16.

[55] Syama Sundar Rangapuram et al. "Deep state space models for time series forecasting". In: *Advances in Neural Information Processing Systems*. 2018, pp. 7785–7794.

[56] Rob J Hyndman, Yeasmin Khandakar, et al. *Automatic time series for forecasting: the forecast package for R*. 6/07. Monash University, Department of Econometrics and Business Statistics …, 2007.

[57] Di Chai, Leye Wang, and Qiang Yang. "Bike flow prediction with multi-graph convolutional networks". In: *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM. 2018, pp. 397–400.

[58] Kunjin Chen et al. "Short-term load forecasting with deep residual networks". In: *IEEE Transactions on Smart Grid* (2018).

[59] Filipe Rodrigues, Ioulia Markou, and Francisco C Pereira. "Combining time-series and textual data for taxi demand prediction in event areas: A deep learning approach". In: *Information Fusion* 49 (2019), pp. 120–129.

[60] Lei Lin, Zhengbing He, and Srinivas Peeta. "Predicting station-level hourly demand in a large-scale bike-sharing network: A graph convolutional neural network approach". In: *Transportation Research Part C: Emerging Technologies* 97 (2018), pp. 258–276.

[61] Jun Xu et al. "Real-time prediction of taxi demand using recurrent neural networks". In: *IEEE Transactions on Intelligent Transportation Systems* 19.8 (2017), pp. 2572–2581.

[62] Chih-Hsin Chou et al. “Long-Term Traffic Time Prediction Using Deep Learning with Integration of Weather Effect”. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2019, pp. 123–135.

[63] Lei Lin et al. “Quantifying uncertainty in short-term traffic prediction and its application to optimal staffing plan development”. In: *Transportation Research Part C: Emerging Technologies* 92 (2018), pp. 323–348.

[64] Hoang Nguyen et al. “Deep learning methods in transportation domain: a review”. In: *IET Intelligent Transport Systems* 12.9 (2018), pp. 998–1004.

[65] Loïc Bontemps, James McDermott, Nhien-An Le-Khac, et al. “Collective anomaly detection based on long short-term memory recurrent neural networks”. In: *International Conference on Future Data and Security Engineering*. Springer. 2016, pp. 141–152.

[66] Subutai Ahmad et al. “Unsupervised real-time anomaly detection for streaming data”. In: *Neurocomputing* 262 (2017), pp. 134–147.

[67] *Numenta*. URL: `https://numenta.org/hierarchical-temporal-memory/`.

[68] Kyle Hundman et al. “Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2018, pp. 387–395.

[69] Erik Marchi et al. “Non-linear prediction with LSTM recurrent neural networks for acoustic novelty detection”. In: *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2015, pp. 1–7.

[70] Sucheta Chauhan and Lovekesh Vig. “Anomaly detection in ECG time signals via deep long short-term memory networks”. In: *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2015, pp. 1–7.

[71] Hari Mohan Rai, Anurag Trivedi, and Shailja Shukla. “ECG signal processing for abnormalities detection using multi-resolution wavelet transform and Artificial Neural Network classifier”. In: *Measurement* 46.9 (2013), pp. 3238–3246.

[72] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhudinov. "Unsupervised learning of video representations using lstms". In: *International conference on machine learning*. 2015, pp. 843–852.

[73] Jason Brownlee. *4 Common Machine Learning Data Transforms for Time Series Forecasting*. 2018. URL: https://machinelearningmastery.com/machine-learning-data-transforms-for-time-series-forecasting/.

[74] *Hopsworks*. URL: https://github.com/logicalclocks/hopsworks.

[75] *Logical Clocks*. URL: https://www.logicalclocks.com/.

Appendix A

Autoencoder reconstruction for all 28 timesteps of a time series

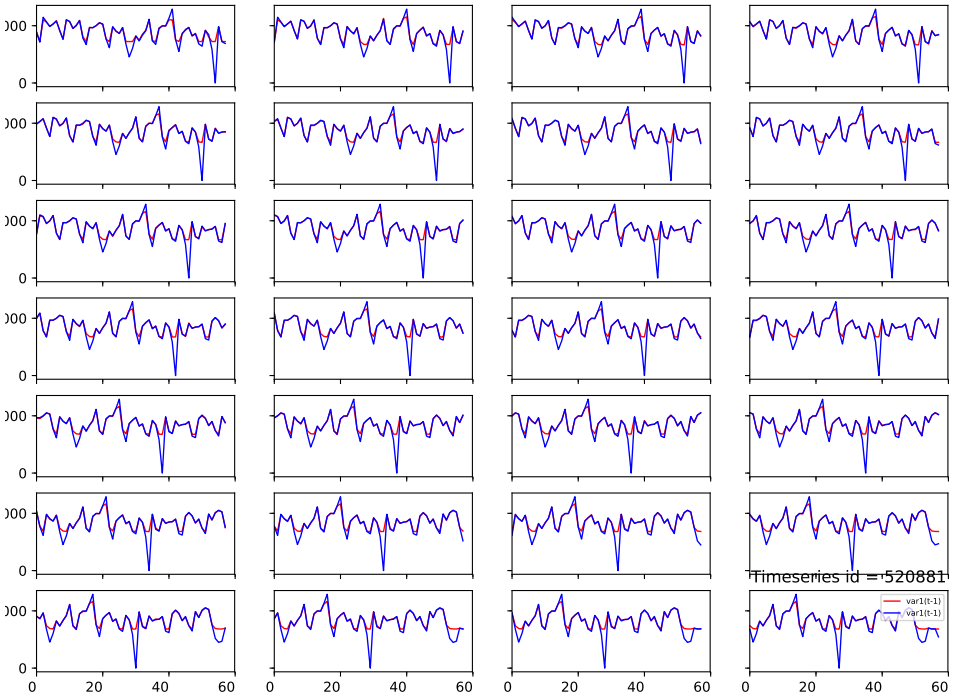


Figure A.1: AE reconstruction for all 28 timesteps of time series with id 520881

Appendix B

Uncertainty estimation

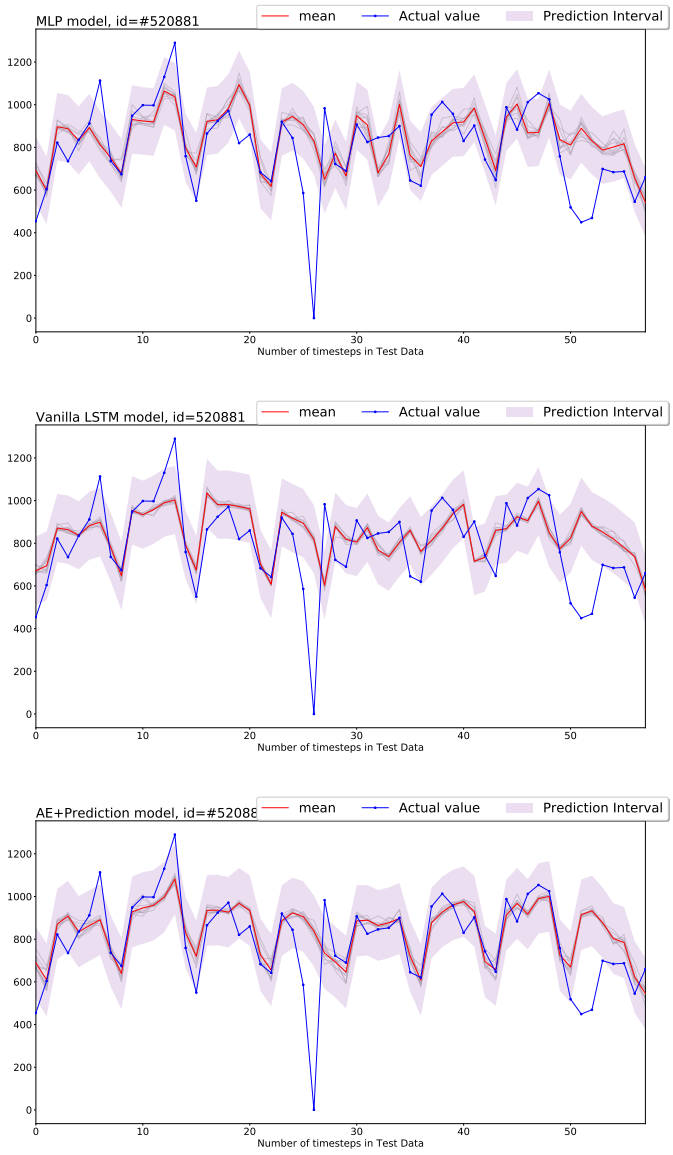


Figure B.1: Uncertainty estimation for three models for time series with ID=520881

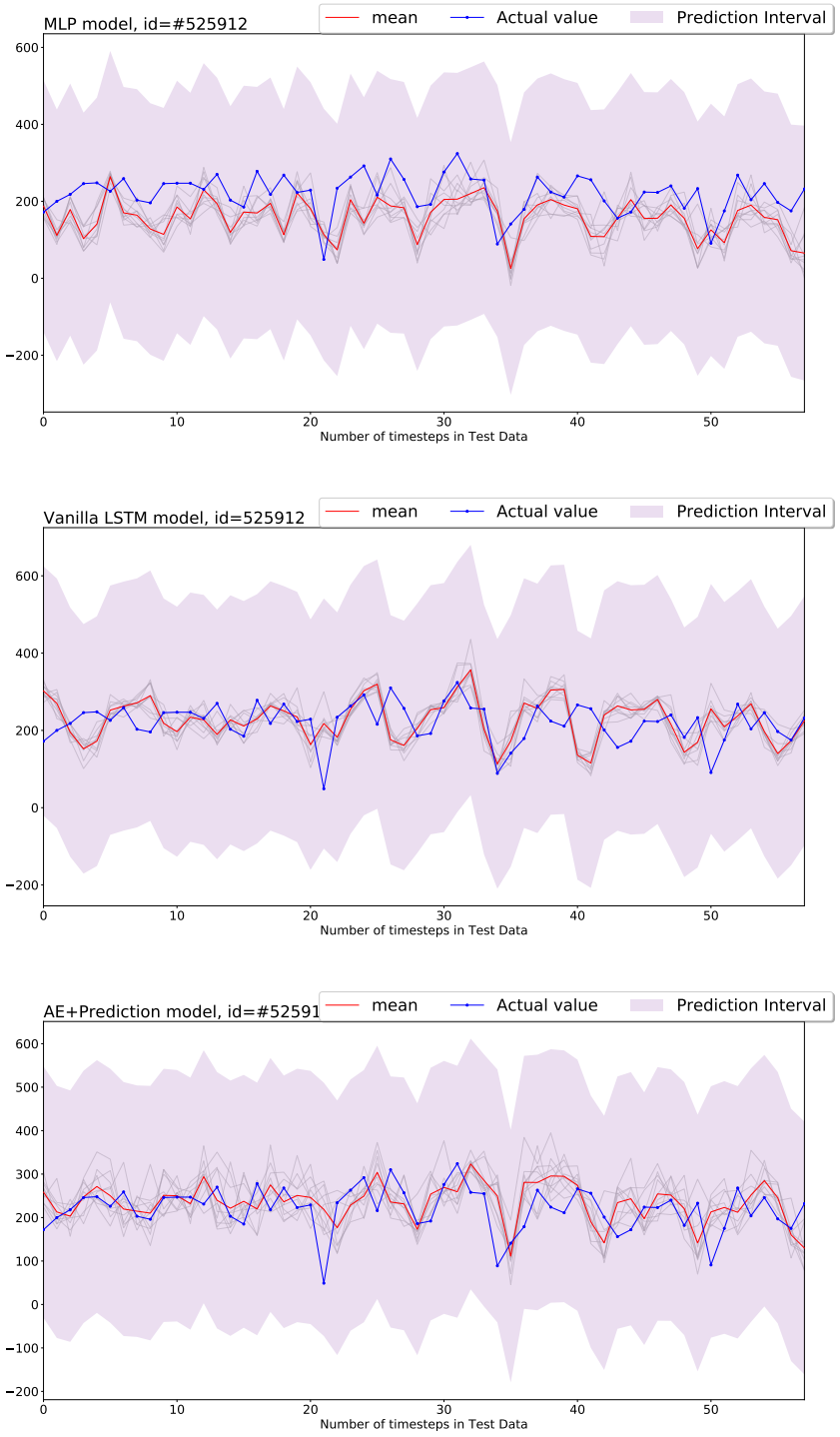


Figure B.2: Uncertainty estimation for three models for time series with ID=525912

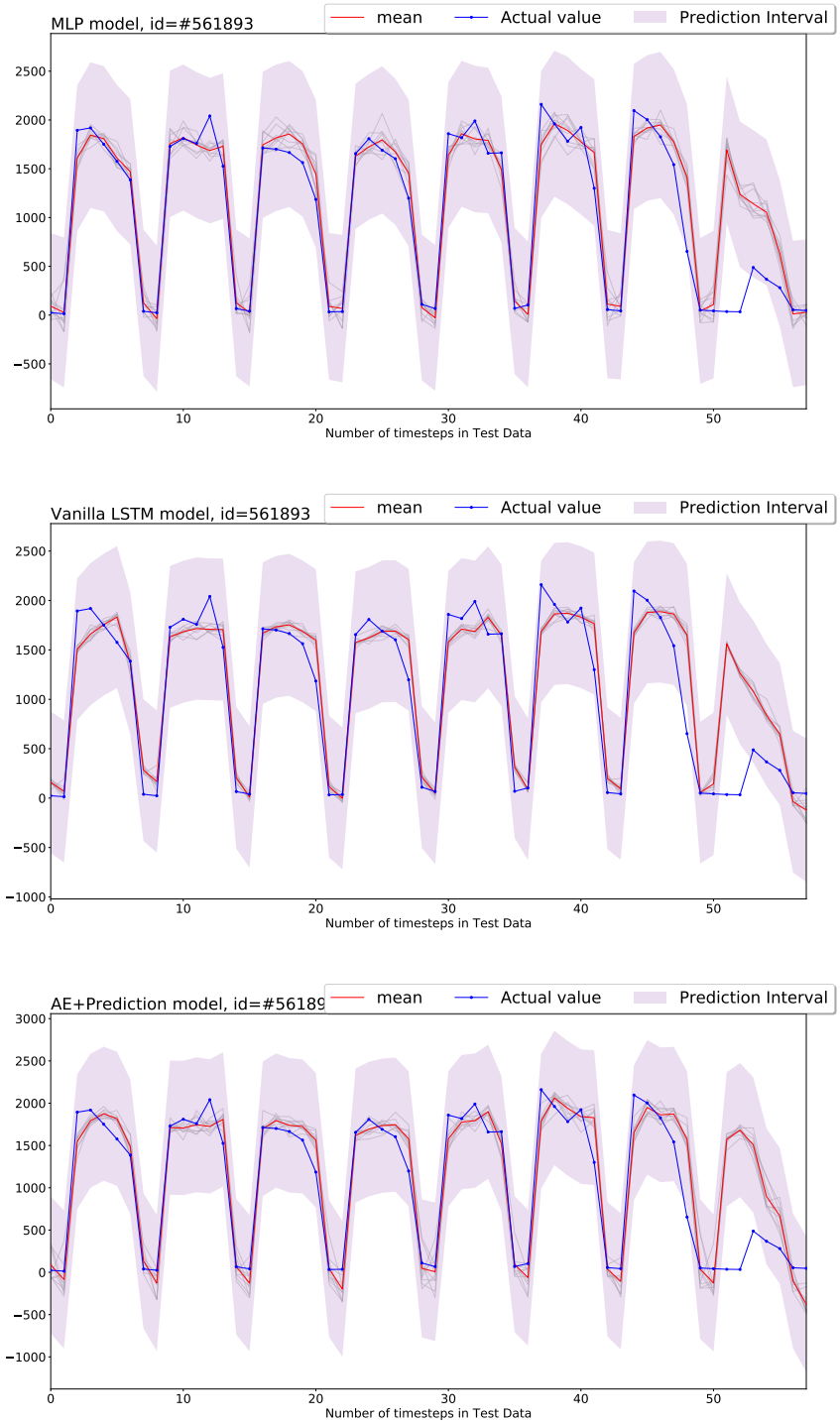
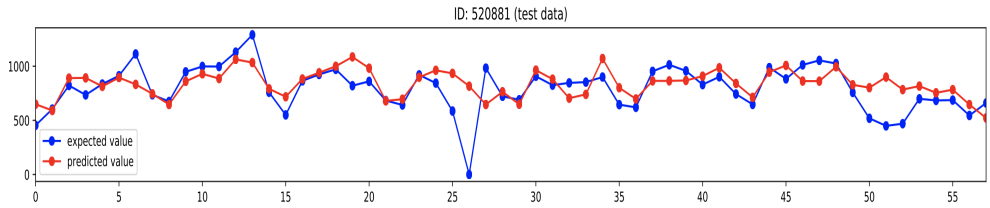


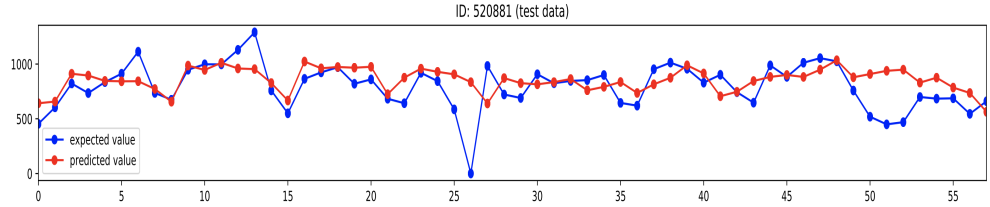
Figure B.3: Uncertainty estimation for three models for time series with ID=561893

Appendix C

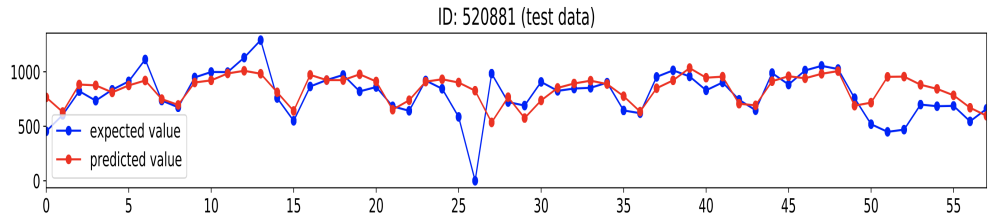
Comparing prediction results



(a) MLP model

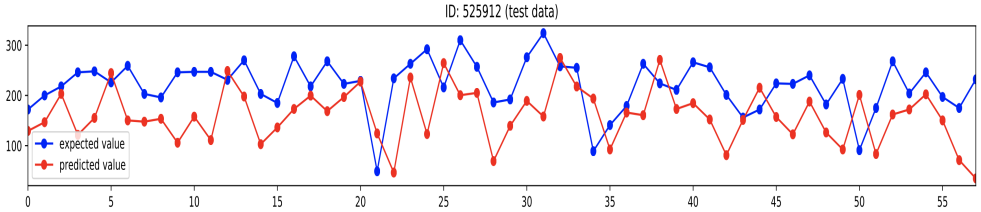


(b) Vanilla LSTM model

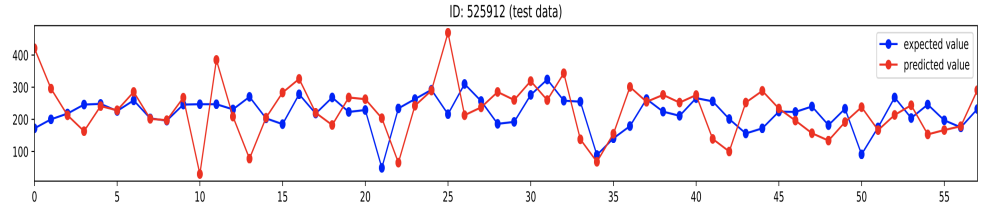


(c) AE+Prediction model

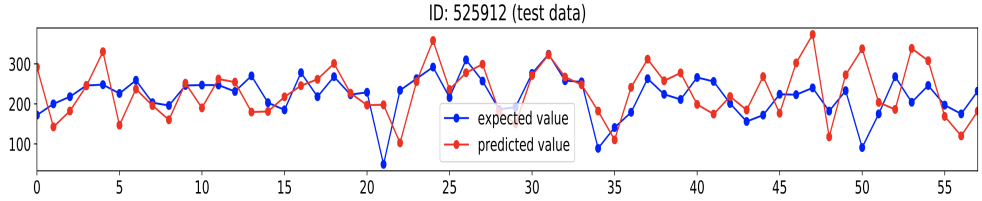
(d) Prediction results of three models for time series with ID=520881



(a) MLP model

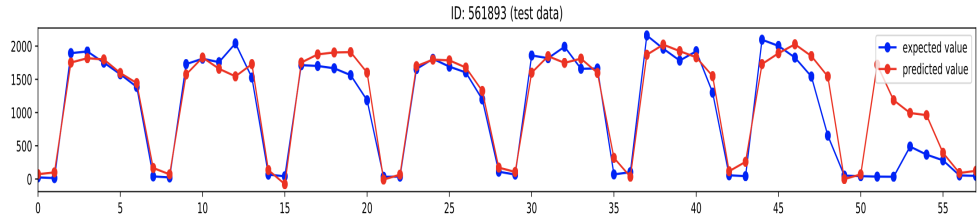


(b) Vanilla LSTM model

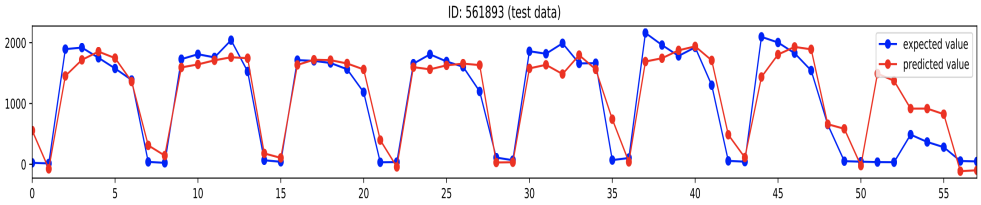


(c) AE+Prediction model

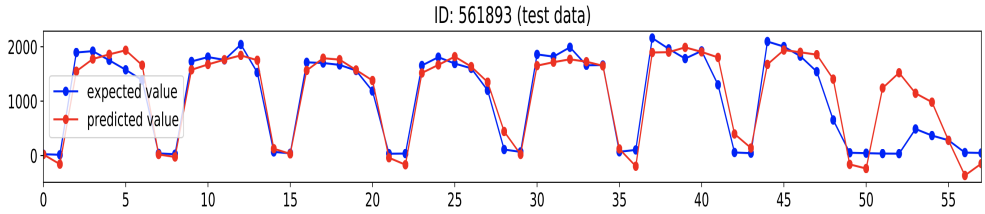
(d) Prediction results of three models for time series with ID=525912



(a) MLP model



(b) Vanilla LSTM model



(c) AE+Prediction model

(d) Prediction results of three models for time series with ID=561893

TRITA EECS-EX