Master's thesis 2021

Sequential Anomaly Detection for Log Data Using Deep Learning

LINA HAMMARGREN
WEI WU

Abstract

Software development with continuous integration changes needs frequent testing for assessment. Analyzing the test output manually is time-consuming and automating this process could be beneficial to an organization. The goal of this thesis project is to do the automated anomaly detection analysis of software test output files provided by Volvo Group Trucks Technology, to achieve this we evaluated four different neural network architectures. The four neural network architectures are two recurrent neural networks with long short-term memory (LSTM) where one is unidirectional and one is bidirectional as well as two autoencoders (an LSTM-based sequence-to-sequence model and a Transformer) that aim to reconstruct a sequence from the files.

In order to evaluate the performance of the neural network architectures two datasets were utilized. The first dataset is from the Hadoop Distributed File System (HDFS) and this is a publicly available dataset where all logs are labelled as either anomalous or non-anomalous. The second dataset are log files resulting from software testing provided by Volvo Group Trucks Technology which contain no labels. The networks were evaluated in two different settings when trained on the HDFS data. In the first setting the logs labelled as anomalous were filtered out making it a *semi-supervised* approach and in the second setting the logs labelled as anomalous were kept which makes it an *unsupervised* approach. Lastly the networks were trained on the data provided by Volvo Group Trucks Technology which is unlabeled, the objective of approach is to evaluate how the networks perform in an unsupervised setting. In addition, an analysis of the size of the data sets used to train the networks were performed.

The results show that for the data provided by Volvo Group Trucks Technology the size of the dataset used for training the networks influenced the performance of the anomaly detection where a smaller dataset performed better than a larger dataset. Moving on to the HDFS dataset, a smaller dataset for the unsupervised setting was also better than a larger dataset. However, for the HDFS data the semi-supervised approach outperformed the unsupervised setting regardless of the size of the training dataset.

Keywords: anomaly detection, recurrent neural network, long short-term memory, semi-supervised learning, seq2seq, transformer, unsupervised learning, log analysis.

Acknowledgements

First, we would like to express our special appreciation of gratitude to our supervisor Anton Johansson at the University of Gothenburg as well as our supervisor Fanny Sandblom at Volvo Group Trucks Technology who have provided guidance in the implementation details of the neural network methods and knowledge of the data. Additionally, we would like to thank the manager at Volvo Group Trucks Technology, Magnus Ståhlesjö, who gave us the opportunity to do this project. We also want to thank Dhasarathy Parthasarathy who continuously provided us with input and lended us a GPU to assist in the training of the neural networks.

Secondly, we would also like to thank each other for support and collaboration in completing the thesis project. A special acknowledgement we would also like to make to LogPAI [17] which was used as a starting ground for research about anomaly detection methods in computer log data.

Finally, we would like to thank our examiner Serik Sagitov at the University of Gothenburg for valuable input to produce the final version of this thesis. Without everyone involved, we would not have finished this project within the limited time frame.

Lina and Wei, Gothenburg, June 2021

Contributions

Both of us have read through the thesis and contributed to the final version. However, the work of writing have been divided so that one of the authors have main responsibility for each section. We will start with presenting the main authors of each of the sections in the written thesis report.

Introduction (Chapter 1): Lina Hammargren

Theory (Chapter 2): Wei Wu

Method (Chapter 3): Lina Hammargren

Results (Chapter 4):
Section 4.2: Lina Hammargren
Section 4.3: Wei Wu

Discussion and conclusion (Chapter 5):
Section 5.1: Wei Wu and Lina Hammargren
Section 5.2: Lina Hammargren
Section 5.3: Wei Wu and Lina Hammargren

Regarding the implementation work we divided the work so that Wei Wu has been responsible for producing the results for the HDFS dataset and Lina Hammargren has been responsible for producing the results for the Volvo GTT dataset. Wei Wu has programmed the methods using Keras and Lina Hammargren has programmed the methods using Pytorch.

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

1.1.1 Logs in computer systems

Logs are reports of the behaviour of a software system during runtime [32]. The logs are a collection of logging statements entered into the code by developers of the system and used for the evaluation of the performance of the software. For instance, a logging statement could describe an error in the code or the operation performed by the software.

Software testing aims, as the name suggests, to test how a program performs. The software test output used in this project is a result of regression tests performed once a day with continuous integration of changes. The software testing results are in the form of log files containing the completed commands and the corresponding result. The log files obtained from this process are then analysed offline, meaning that they are analyzed after the test has been performed and the log file has been created. Of special interest are the irregular behaviours of the software, the anomalies. However, the log files contain a lot of information and large quantities of log files are produced as a result of the regression tests. Looking for anomalies manually in all produced log files is therefore time-consuming and automating this process could potentially speed up the detection of anomalies.

1.1.2 Automated anomaly detection in log data

Automated log anomaly detection is a research field that concerns methods of identifying irregularities in the log data without human intervention. In the field, there are various proposed machine learning methods using for example clustering techniques [28] or Principal Component Analysis (PCA) based methods [33]. Deep learning [23] methods have shown promising results in the area, several of them [7, 24, 35] utilizing recurrent neural networks (RNN).

This project will focus on sequence modelling of the log files for anomaly detection using deep learning. Most research in RNN:s for automated log anomaly detection [7, 35, 26] use a semi-supervised approach which in this case means that only logs without sequential irregularities, or 'normal' execution path data, are used to train the models. In this project we will mainly use an unsupervised approach and test the assumption that the proportion of anomalies is small and in turn the model will not be able to learn the patterns of these instances.

In order to evaluate the methods we will use two data sets. The first one consists of the software test output from Volvo Group Trucks Technology (Volvo GTT) which contain no information regarding which logs are anomalies except for a small data set containing labelled log files used for evaluation. The second data set is from the Hadoop Distributed File System (HDFS) and is a publicly available data set [13] containing labels on a log level. The methods will be evaluated on the Volvo GTT data set, the HDFS data set containing both anomalies and logs with normal behaviour and lastly the HDFS data set with the anomalous logs filtered out.

1.1.3 What is considered an anomaly?

Before giving an intuition behind what an anomaly could be in the data we need to explain a bit about *logging statements*. Logging statements are messages inserted into the code by developers intended to be printed out as a program is running. In the following text logging statements will also be refered to as *log lines*.

One logging statement can be divided into *variable* and *static* components. The variable components of a logging statement is the information that may change for different runs of the program. For example, the timestamp in a logging statement will change if a program is run at two different points in time. In order to explain the variable components a written explanation is usually entered into the logging statement to make the logs more comprehensible and this is called the static component. Figure 1.1 contains a simple fabricated logging statement where the variable and static component has been pointed out.

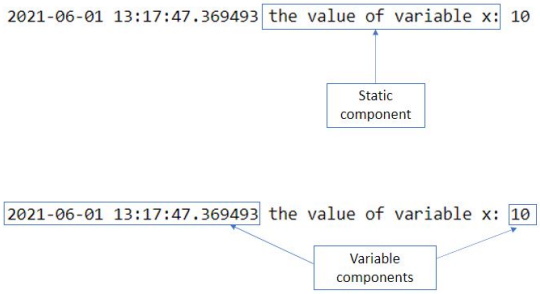

Figure 1.1: The static and variable components of a fabricated logging statement.

In this project we will model the logs as sequences of *events* defined by the logging statements in a log. The events will in this project be defined as the *static component* of a logging statement. An anomaly is then the pattern of a log corresponding to a faulty process. Figure 1.2 depicts a simple fabricated example of a process working as intended and figure 1.3 a faulty process. However, in the log data that has been used in this project the anomalies might not be as easy to detect and may for example consist of several lines which is why a deep learning approach was tested.

```
2021-06-01 16:48:35.800657 program has started
2021-06-01 16:48:35.800657 the value of variable x: 64
2021-06-01 16:48:35.800657 the value of variable x: 27
2021-06-01 16:48:35.800657 the value of variable x: 36
2021-06-01 16:48:35.800657 the value of variable x: 7
2021-06-01 16:48:35.800657 the value of variable x: 96
2021-06-01 16:48:35.800657 the value of variable x: 97
2021-06-01 16:48:35.800657 the value of variable x: 15
2021-06-01 16:48:35.800657 the value of variable x: 65
2021-06-01 16:48:35.801654 the value of variable x: 69
2021-06-01 16:48:35.801654 the value of variable x: 41
2021-06-01 16:48:35.801654 task completed successfully
2021-06-01 16:48:35.801654 start disconnecting
2021-06-01 16:48:35.801654 disconnect successful
```

Figure 1.2: Fabricated example of a normal log with the intended outcome.

```
2021-06-01 16:46:33.082565 program has started
2021-06-01 16:46:33.082565 the value of variable x: 91
2021-06-01 16:46:33.082565 the value of variable x: 15
2021-06-01 16:46:33.083580 the value of variable x: 80
2021-06-01 16:46:33.083580 the value of variable x: 45
2021-06-01 16:46:33.083580 the value of variable x: 81
2021-06-01 16:46:33.083580 the value of variable x: 70
2021-06-01 16:46:33.083580 the value of variable x: 92
2021-06-01 16:46:33.083580 the value of variable x: 35
2021-06-01 16:46:33.083580 the value of variable x: 97
2021-06-01 16:46:33.083580 the value of variable x: 9
2021-06-01 16:46:33.083580 the value of variable x is out of range
2021-06-01 16:46:33.083580 start disconnecting
2021-06-01 16:46:33.083580 disconnect successful
```

Figure 1.3: Fabricated example of an anomalous log with a faulty process.

1.2 Overview of Related Work in Automated Anomaly Detection for Logs

In order to investigate research that has been done in the field of log anomaly detection LogPAI [17] has been used as a source of information. LogPAI is an open-source platform containing research and implementation details for automated log analysis.

The focus of this thesis has been on approaches based on neural networks and there are several works presenting neural network-based approaches to automated anomaly detection. One possible approach is to use a recurrent neural network architecture with long short term memory (LSTM) cells [7, 24] where the neural network model is taught to predict the most likely subsequent log key given some past context from the log sequence. This can be used for anomaly detection by pointing out predictions that are not among the most probable subsequent log keys as anomalies. These types of networks can also be used with attention mechanisms, such as in [35].

Other neural network-based approaches focus on the vector representations of logs to be able to detect anomalous log lines as numerical outliers. For example, [26] used a Transformer encoder model with the objective to enforce normal log lines to be close to each other in vector space forming a cluster while the anomalous log lines are further from this cluster of normal log lines. This is done to improve the performance of clustering methods. Such an approach would require information regarding which logs are considered non-anomalous and can therefore not be applied to the data provided by Volvo Group Trucks Technology (Volvo GTT).

Anomaly detection can also be implemented through an autoencoder neural network. In this approach the objective is to reconstruct the initial input to the network. Autoencoders built as a sequence-to-sequence network with LSTM recurrent neural networks have been used to reconstruct time series [22] to name an example. Additionally, sequence-to-sequence networks are used in machine translation [30, 31]. In this project the data consist of integer sequences corresponding to a temporally ordered sequence of log events, so instead of reconstructing some numerical vector we aim to predict the integer (log event class) at each time instance. In short, each sequence input is fed to an encoder and a representation of the original sequence is fed to a decoder which then reconstructs the log sequence by predicting each log key at the respective temporal position.

Additionally, other networks such as Transformers used for neural machine translation have also been used for anomaly detection in logs [15]. Transformers handle sequence data but does not utilize recurrency in the manner that RNN:s do and have been shown to perform better than RNN:s with LSTM in neural machine translation tasks [31]. In this project a Transformer autoencoder was implemented that aims to reconstruct a log sequence in the same manner as the sequence-to-sequence networks with RNN:s.

Moving back to the data used in this project, we do not have access to a large sample of labeled data for the data provided by Volvo GTT and therefore do not know which logs are anomalous and which ones are non-anomalous during the training of the neural networks. This is different to the approaches in [7, 35] where the recurrent neural networks are trained using data that conforms to normal behaviour. Because of this we replicated our methods on the Hadoop Distributed File System (HDFS) data [13] which has been used in these articles. We replicate our method using both mixed data (data containing both regular behaviour and irregular behaviour) and data only containing logs with only regular behavious.

1.3 Project Description and Aim

The project was made available to us by Volvo Group Trucks Technology (Volvo GTT) with the aim of exploring different data mining methods on their software test output data. The purpose of the project is to find an efficient log analysis method that alleviates the manual log analysis process. We wanted to explore different machine learning methods to obtain this goal but did not have access to a large amount of labeled data. Therefore the main task of this project is to find an anomaly detector that performs well on the Volvo GTT software test output data.

The aim of this project is further to investigate whether it is possible to create an anomaly detector that has been trained in a completely unsupervised manner. To this end we have explored different unsupervised anomaly detection methods on the software test output provided by Volvo GTT. We want to compare and contrast different types of neural network architectures in order to investigate which types of architectures perform the best and potentially find an anomaly detector that can be used in practice. The approach is unsupervised as there are no indications of what could be an abnormal or normal log line or log file during training. Instead, neural network models will be trained to learn the patterns of the sequence data and the log files which the models do not perform well on are defined as predicted anomalies.

The architectures include LSTM-based models trained to predict a subsequent log event, LSTM-based models that aim to reconstruct a sequence and a Transformer architecture that also aim to reconstruct a sequence. The purpose is to find the best performing architecture with corresponding hyperparameters. The hyperparameters we have used are guided by previous work done in the respective domain and research.

To be able to compare our results we utilized a publicly available labelled data set from the Hadoop Distributed File System (HDFS) [13] and replicated the entire procedure used on the Volvo GTT dataset.

To conclude, the main objectives of this project has been summarized project below in (i)-(iv).

(i) Literature study of unsupervised deep learning approaches for anomaly detection

(ii) Investigation of different neural network approaches

(iii) Implementation and comparison of different approaches on the Volvo GTT dataset and the HDFS dataset

(iv) Evaluation of whether any of the implemented methods can be used in an unsupervised setting to partly automate the log analysis for the Volvo GTT data

1.4 Limitations and Scope of the Project

The anomaly detectors for the Volvo GTT data has been trained with data where there are no indications of what log files could contain anomalies. Therefore the approach is completely unsupervised. Furthermore, as mentioned in Section 1.2 in most other research that we are aware of only log files confirmed as 'non-anomalous' have been used to train the neural networks. For the Volvo GTT data we train the neural networks with log files that could be both anomalous and non-anomalous. The assumption is that the portion of anomalies would be so small that the network should not be able to learn the patterns of these sequences. For comparison to related work we have trained all of our models on both mixed data and only normal data for the HDFS dataset.

During the course of this project we obtained a small labeled data set of log files from the software test jobs to be able to evaluate the anomaly detection methods. In this data set there are a total of 81 log files out of which 8 of these are confirmed anomalous by developers at Volvo GTT and the other 73 log files are assumed to contain no anomalies. In this data set the logs are labeled on a file level as being either anomalous or non-anomalous. This dataset used for anomaly detection is imbalanced and small which is why we used the HDFS dataset which contains more labeled log sequences to get a more reliable result.

The Volvo GTT data set is on a file-level and in each file there are several temporally ordered log lines containing information about the software tests performed. The raw HDFS data set contains only log lines that can be sorted by a special identification called a *block-id*. The block-id is unique for each log sequence so we sorted the raw HDFS data set by grouping together the log lines belonging to a certain block-id. In the HDFS data set we also observed that the length of a log sequence belonging to a certain block-id can be an indicator of whether it is an anomaly or not in itself. This was not observed for the Volvo GTT data and quantitative anomalies are therefore not covered in this project.

Lastly, anomalies in log files can appear in the form of parameter value anomalies such as the time it took to run a command, or abnormal numerical outcomes of a command. In this thesis we focus only on sequential pattern anomaly detection with the text information from logs. The reason for this is that we are starting from a point where no machine learning methods have been tried for this data set. Thus, we made the choice of restricting ourselves to sequential anomaly detection.

1.5 Outline of the Thesis

The thesis starts with an outline of the theoretical aspects of the neural networks utilized in this project in Chapter 2. The chapter starts with a general introduction to neural networks and recurrent neural networks. Also in Chapter 2, there is a detailed background of the long short-term memory (LSTM) cell, how the neural network learns, and the individual components of the neural network architectures.

Chapter 3 will present the data pre-processing and model architectures then end with an explanation of how the methods have been used to perform anomaly detection. Next, Chapter 4 shows all the results of the applied models and methods. Last, Chapter 5 presents a discussion of the results and the conclusions we have drawn from the project and gives suggestions for future work.

2
Theory

Neural network methods are a subset of machine learning methods, and machine learning methods are commonly divided into *supervised* and *unsupervised* learning. In supervised learning such as classification, the target (truth class) is known. In unsupervised learning, it is usually the case that the response (target) variable is unknown. Referring back to chapter 1, we mentioned that we are doing unsupervised anomaly detection. However, in the approaches presented in this project, we utilize entries from the log sequence as targets to be able to use supervised neural network methods. The methods used in this project are unsupervised due to the fact that we do not know which logs are anomalous and non-anomalous in the Volvo GTT dataset during training of the neural networks.

In this chapter, we will present some theoretical background for neural networks. This starts with a general background to feed-forward neural networks in section 2.1 and then move on to recurrent neural networks (RNNs) in section 2.2 which can be used for sequence learning which is the main topic in this project. After the general theory behind RNNs we will in section 2.3 explain how neural networks learn from data. Lastly, in section 2.4 we will present the individual components and their function of the neural networks that will be used in this project.

Mathematical Notation

Notation	
i/j	Index
t	Time step
x	Input
O	Output
y	Target
w	Weight
W	Matrix of weights
b	Bias
h	Hidden state
c	Cell state
K	Amount of classes

2.1 Neural Networks

The architectures of neural network methods are modelled on neurons in the brain [23]. Figure 2.1 presents how one neuron generates output given some input x and is a mathematical simulation of the neuron. The work progress will be like this: the dendrites will carry the signals (the product of the input x and weight w, e.g., w_0x_0, w_1x_1, ...) to the cell body where they all get summed. Bias b is an additional parameter to adjust the output and the weighted sum of the inputs to the neuron and each neuron has its own bias. The signal transferred by the neuron is a function $f()$ called an activation function of the sum $\sum_i w_i x_i + b_i$. Some example activation functions used in this project will be presented in section 2.1.1.

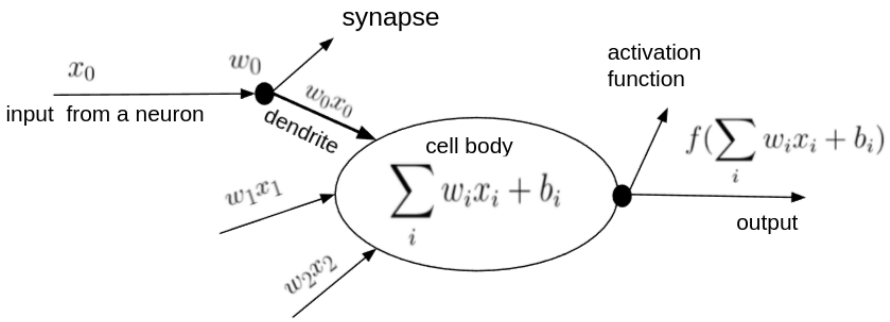

Figure 2.1: The mathematical illustration of one neuron.

In a neural network, the neurons are the smallest computational units which are connected to each other and together build the network. Figure 2.2 shows how an example network is structured with blocks of neurons, also called a layer of neurons. The illustration specifically shows an example of a *feed-forward* neural network with one input layer, one hidden layer, and one output layer. Each neuron corresponds to the individual neuron in Figure 2.1 and has its associated weight. In general, the weights between different pairs of neurons can be both positive or negative and when the weight is zero it means that there is no connection between the pairs of neurons. In addition, the computation is performed for all neurons in parallel, and the outputs can be the inputs to other neurons at the next time step. Looking back at Figure 2.1 the output of the neuron $f(\sum_i w_i x_i + b_i)$ is the input for the next connected neuron.

When the neural network is training, the idea is to let the neuron transfer the input to receive the output. The activation functions $f()$ may be chosen during the construction of the network, and different activation functions will give different types of output. A simple linear model will be obtained if there are no activation functions on any layer. The choice of the activation functions will depend on the situation and we refer to section 2.1.1 for the activation functions used during this project.

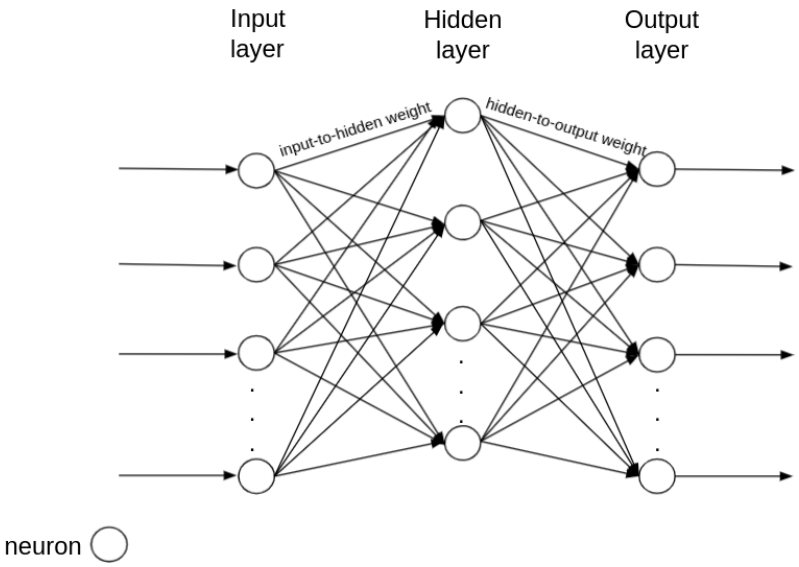

Figure 2.2: Illustration of an example feed-forward neural network.

Forward propagation is when input is fed to the model and the output is received from this input. To name an example, in a classification task the output could be the prediction of which class the input belongs to. In order to allow the network to learn we use backward propagation of errors (backpropagation) where we go in the opposite direction of forward propagation to minimize a *loss function*. The loss function is used to measure the error between the ground truth and the prediction. The goal during the training is to minimize the loss function. We use gradient descent methods to adjust the parameters to minimize the loss, the objective is to calculate the gradients of the loss functions with respect to weights and biases in the neural network and fine-tuning the weights and biases according to the error rate from previous learning (the parameter values of weights and biases for every single neuron can be adjustable).

Different types and amounts of layers will form different types of neural networks. See Figure 2.2 again, in this example neural network, only one hidden layer exists. If there exist more hidden layers, this is a deep neural network. However, it is not correct that choosing more hidden layers will always get better results. The complexity of the data will determine the amount of hidden layers and widths of these since if the choice of the amount of layers is not correct, the model can either underfit or overfit. Generally, underfitting is the case where the model has "not learned enough" from the training data, resulting in bad predictions on both the training data and unseen data, and overfitting is the case where the model has "learned too much" from the training data, resulting in good predictions on the training data but bad predictions on the unseen data.

There are several common neural networks such as single-layer perceptron, convolutional neural network, and recurrent neural networks.[23] In section 2.2, the

recurrent neural network will be explained in detail since it will appear frequently in the project.

2.1.1 Activation functions

As mentioned before, the purpose of the activation function is to enable the network to learn complex patterns, and it will decide what to be transfer to the next neurons. Here we outline three activation functions, sigmoid, tanh and ReLU, that will be used in the architectures presented in this project. The output range of the sigmoid function is 0 to 1; the output range of the tanh function is -1 to 1; and the range of the ReLu function is 0 to infinity. Figure 2.3 to 2.5 shows the illustration of the functions:

Sigmoid function (σ)

$$\sigma(b) = \frac{1}{1 + e^{-b}} \tag{2.1}$$

Tanh function (tanh)

$$\tanh(b) = \tanh(b) \tag{2.2}$$

ReLU function ($ReLU$)

$$R(b) = max(0, b) \tag{2.3}$$

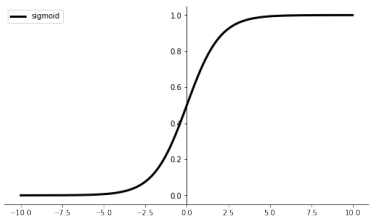

Figure 2.3: Sigmoid activation function.

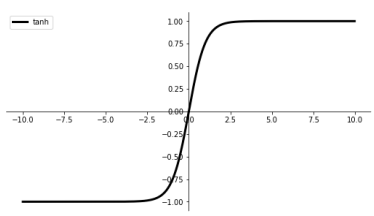

Figure 2.4: Tanh activation function.

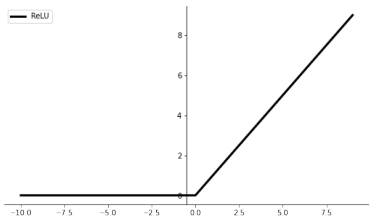

Figure 2.5: ReLU activation function.

2.2 Recurrent Neural Networks

In the previous section Figure 2.2 showed an example of a feed-forward network. In a feed-forward network the neurons are connected to each other in a forward fashion. Recurrent neural networks (RNN) are neural networks with feedback loops, as depicted in Figure 2.6. The feedbacks here can be placed in different ways: from the output layer to the hidden neurons for example, or the connections between the neurons in given layers. As shown in Figure 2.6: x_1 and x_2 are the input , h_1 is the hidden neuron. O_1 and O_2 are the output neurons. The weight from the input x to the hidden neurons h is denoted as w^{xh}.

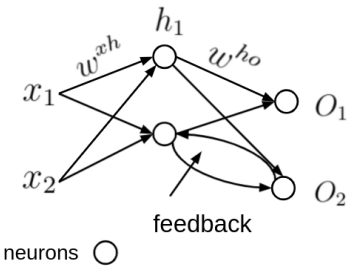

Figure 2.6: Neural network with a feedback connection.

Recurrent neural networks handle sequential data and introduce the concept of a time step. A time step is the part of a transfer in which neuron inputs are processed into outputs, and then those outputs are fed to the next neuron. Basically, the amount of time steps is equal to the length of the input, and it depends on the setup: it can be characters, words or items. For example, if we feed the data word "math" to the network, the input at time step 1 will be the character "m" and the input at time step 2 will be the character "a" and so on.

In Figure 2.7 the evolution of a recurrent neural network over time is presented, the left part refers to a single cell of the recurrent network and the right is the same network as the left but unfolded in time. The unfolded network has t inputs and outputs (t in here is the amount of time steps), and the weights between the different time steps remain the same. $h(t)$ and $O(t)$ are the hidden state and output at the same time step t, $w^{(hh)}$ is the hidden-to-hidden weight, $w^{(xh)}$ is the input-to-hidden weight, and $w^{(hO)}$ is the hidden-to-output weight.

More specifically, the recurrence in the network comes from performing the same computation for every element in the input. The equations 2.4 and 2.5 will explain how a recurrent neural network evolves over time:

$$h(t) = \sigma(w^{(hh)}h(t-1) + w^{(xh)}x(t) + b) \tag{2.4}$$

$$O(t) = \sigma(w^{(hO)}h(t)) \tag{2.5}$$

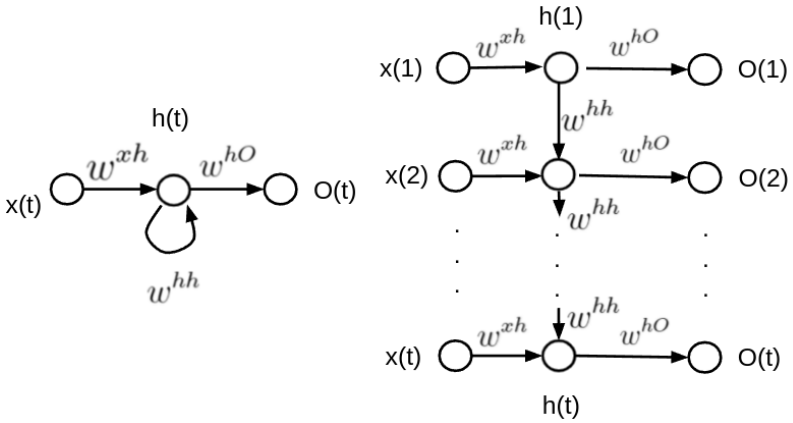

Figure 2.7: A single recurrent network cell (left) and the corresponding recurrent network cell unfolded in time from time step 1 to time step t (right). [34]

As said previously in section 2.1.1, the activation function defined the output of a neural given inputs. $\sigma()$ function is the sigmoid activation function and the output will be a value between 0 and 1.

We mentioned backpropagation which is the process of computing the derivatives of the loss function with respect to the network weights and biases in section 2.1. In a recurrent neural network we backpropagate through both time and layers. In each time step we sum up all the previous contributions until the current one. This computation is presented in (2.6) where the contribution of a state at current time step t' to the gradient of the entire loss function $\mathbf{L}$, at time step t is calculated.

$$\frac{\partial \mathbf{L}}{\partial w^{hh}} = \sum_{i=0}^{t} \frac{\partial Loss(t)}{\partial w^{hh}} \propto \sum_{i=0}^{t} \left(\prod_{i=t'+1}^{t} \frac{\partial h(t'+1)}{\partial h(t')} \right) \frac{\partial h(t')}{\partial w^{hh}} \tag{2.6}$$

Take a look at the ratio of the hidden state, if $\frac{\partial h(t'+1)}{\partial h(t')}$ is less than 1, the equation 2.6 goes to zero exponentially fast, and the training will converge due to this.

The backpropagation computations will be performed more times in the deep recurrent neural network (since there exist more hidden layers in the recurrent neural network). The weights are assigned by sampling from a Normal or Uniform distribution at the start of the neural network, and from there, they are updated during the training process. When the weight of recurrent becomes very small due to repeat multiplication the result can be a vanishing gradient, which may cause the neural network to stop further training of the network completely, which is the vanishing gradient problem.

More specifically, the problem relates to updating the recurring weight – the weight that is used to connect the hidden layers to themselves in the unrolled loop. This can also be seen from Figure 2.7 (see the right part of the figure, the recurring weight here is w^{hh}). To get from $x(3)$ to $x(2)$ we multiply $x(3)$ by w^{hh}. Then, to get from $x(2)$ to $x(1)$ we again multiply $x(2)$ by w^{hh}. So, we multiply with the same weight multiple times, and this is where the problem arises. When multiplying a small value several times, the value decreases very quickly, and can become close to zero. The lower the gradient is, the smaller the update of the weights and biases and the longer it takes to get to the final result, this is called the vanishing gradient problem.

The long short-term memory (LSTM) cell is one approach to solve the vanishing gradient problem [1], the content will be described in detail in Section 2.2.1 below.

2.2.1 Long Short-Term Memory (LSTM)

Long short-term memory (LSTM) is a type of recurrent neural network. The only difference is replacing the hidden neurons of the recurrent network with computation units, which are also called LSTM cells. Figure 2.8 shows the structure of an LSTM network, similar to the recurrent neural network structure but replacing hidden neurons with LSTM cells. The explanatory note on this figure provides two new definitions: cell state and hidden state. The cell state encodes the aggregation of data from all previous time steps that have been processed, and the hidden state encodes the characterization data of the previous time step. The time step t is the current time, time step $t-1$ is the past time, and time step $t+1$ is the future time.

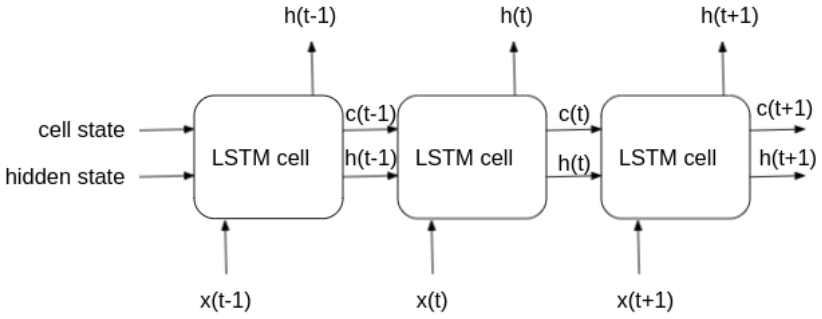

Figure 2.8: The repeating module in an LSTM .

Every LSTM cell has four single neural network layers that interact in some particular way, and they are called the input gate, the forget gate and the output gate. Those gates are internal mechanisms to regulate the flow of information. The main idea to solve the vanishing-gradient problem is to avoid the behaviour of recurring computations. The forget gate in the LSTM cell can let the network encourage desired behaviour from the error gradient using the gates update on every time step. Simply said, the vanishing gradient problem occurs because the weights are multiplied by themselves several times, the forget gates in the LSTM cell will reset the

update to reduce the issue of the repeated multiplication. To understand the exact details, we should look at what happens in the forget gate.

Before the description of the forget gate we present an overview of the whole structure of a single LSTM cell in Figure 2.9. The rectangle with "sig" is the sigmoid activation function (equation 2.1) and the rectangle with "tanh" is the tanh activation function (equation 2.2). Each circle in this figure is an individual neuron. The "+" and "×" inside of the circles represent operations, "+" is vector addition and "×" is component-wise vector multiplication. And one characteristic of LSTM can be seen from the figure: the output will depend on three inputs: $h(t-1)$, $x(t)$ and $c(t-1)$. The $h(t)$ are both the output of the LSTM cell at time step t and a part of the input of the LSTM cell at time step $t+1$.

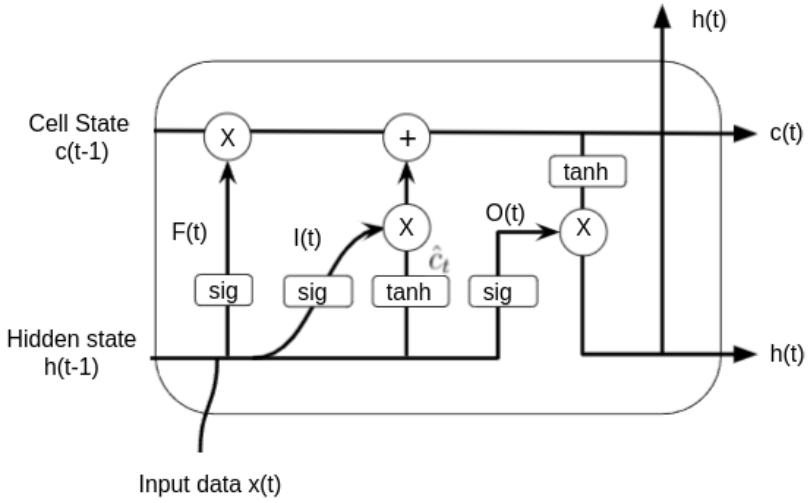

Figure 2.9: The structure of a single LSTM cell.

We start with explaining the forget gate ($F(t)$ in Figure 2.9). The forget gate is the first computational step in the LSTM cell, which can be seen in Figure 2.10 in the single LSTM cell (assume this is not the first or last LSTM cell, it has both the past state and future state). The concatenation of input data from times step t and the hidden state of the past time $h(t-1)$ will be fed through a layer with a sigmoid activation function: the grey area is what is called the forget gate which is the part that decides what information will "forget". The sigmoid activation function outputs a number between 0 and 1. The number 1 here represents "keep all", while the number 0 represents "throw all", a value between 0 and 1 represents the throw percent.

Every gate has entirely separate sets of weights and bias. The function of the forget gate $F(t)$ will be:

$$F(t) = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.7}$$

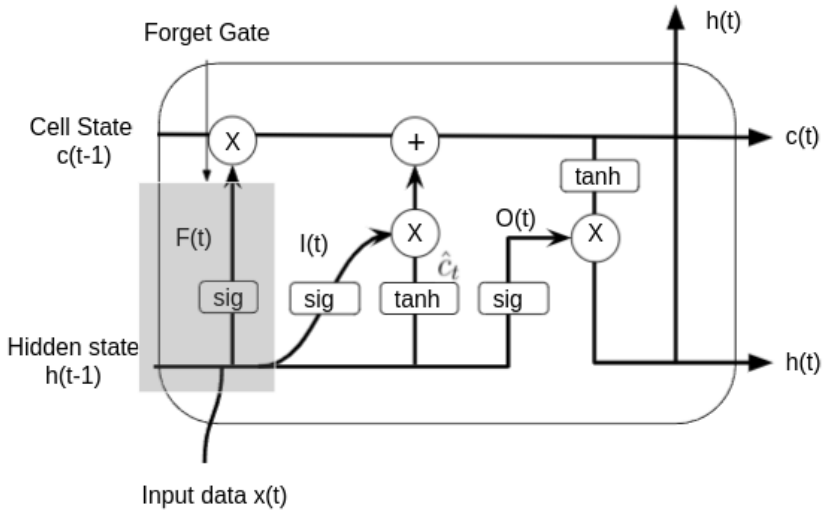

Figure 2.10: Forget gate in LSTM.

This function can be easily seen from the figure,$[h_{t-1}, x_t]$ is the vector concatenation of h_{t-1} and x_t. The weights in the forget gate W_f will determine which time-steps are important (high forget weights), which are not (low forget weights) and encode information from the current time-step into the cell state, then the cell state will be $F(t) * c(t-1)$. If $F(t)$ is zero, the cell state here will also be zero. Then the cell state is reset. This is how the forget gate solve the vanishing gradient problem, once the cell state is reset, the recurring behaviour will be stopped.

Next step is the input gate. See Figure 2.11 where the gray area covers the process of the input gate.

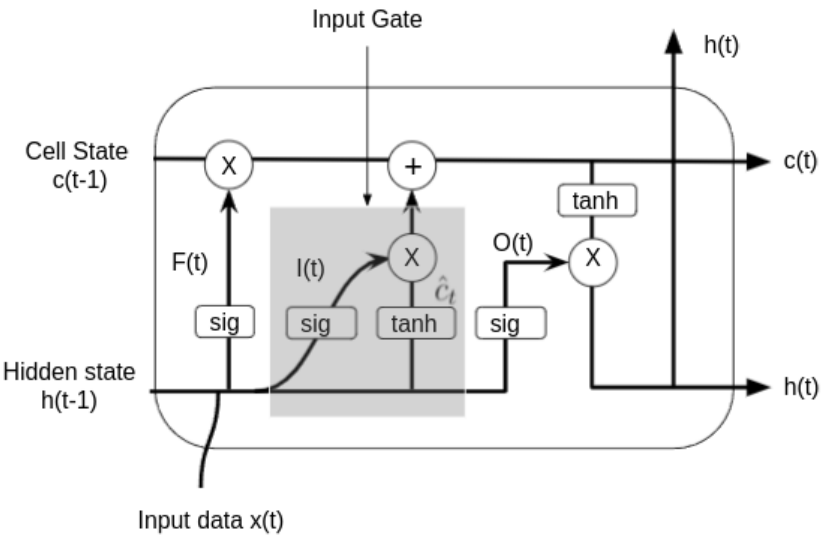

Figure 2.11: Input gate in LSTM.

The input gate will do two functions, one is the same function as forget gate but with the input weights and bias:

$$I(t) = \sigma(W_I \cdot [h_{t-1}, x_t] + b_I) \tag{2.8}$$

This function (equation 2.8) will decide the input values, that is why this gate is named input gate. Another function is to create new values $\hat{c}_t$ (with its own weights and bias) and then through a tanh activation function, which means the range of the values will be between -1 to 1:

$$\hat{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.9}$$

Next combine the $I(t)$ and $\hat{c}_t$ to through the tanh activation function to update the cell state, the values to the cell state will be $\tanh(I_t * \hat{c}_t)$.

Lastly, the gray area of Figure 2.12 shows the diagram of the output gate. In this gate, a sigmoid activation function will decide the values to output by:

$$O(t) = \sigma(W_O \cdot [h_{t-1}, x_t] + b_O) \tag{2.10}$$

Then to decide the values to output in the cell state:

$$c_t = F(t) * c_{t-1} + I_t * \hat{c}_t \tag{2.11}$$

Finally the output $h(t)$ is the filtered version (cell state through the tanh activation function) of cell state $c(t)$ multiplies $O(t)$, and output $h(t)$ is also the hidden state to the time step $t + 1$:

$$h_t = O_t * \tanh(c_t) \tag{2.12}$$

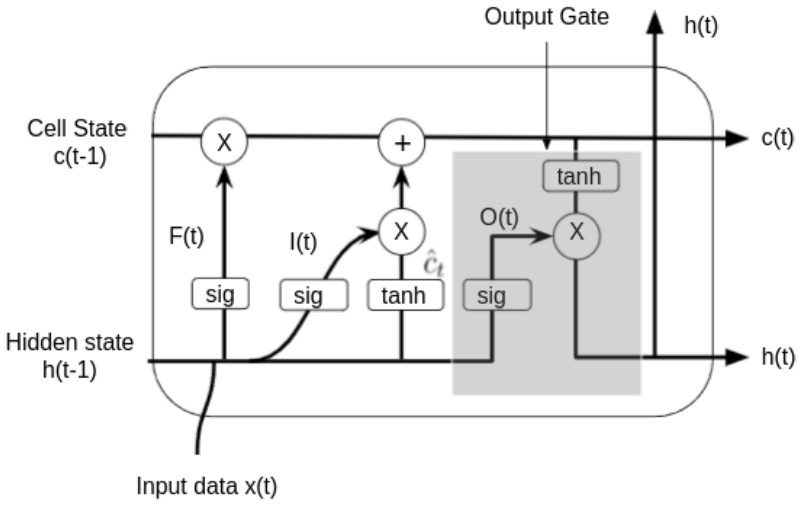

Figure 2.12: Output gate in LSTM.

All information in this section explained how the LSTM works. As the earlier Figure 2.2 shows, the neural network has some layers. The below section will present the process during training the networks.

2.3 Training the Networks

The objective of training a neural network is to minimize the loss function and this is how the neural network "learns". A neural network learns through the forward- and backpropagation algorithms, as mentioned in Section 2.1. The input data is sent to the network architectures in steps and forward propagation is performed to yield the loss value. Then we backpropagate the loss function to obtain the gradients with respect to each changeable parameter. There are different ways of updating the parameters, one of them being the Adam optimizer [16] which adjusts the learning during training.

Before training a neural network model the data is split into training and validation data sets. The network will be learning from the training data and the validation data is not used for learning but only for validating the progress of the training on an unseen dataset (why we need validation data will be explained in the next paragraph). The two datasets are completely independent. During training, the loss and accuracy are evaluated for each *epoch* on the training data set and the validation data set. An epoch is a training instance in which the network has been exposed to the entire dataset once through forward- and backpropagation.

When training a network we want to find the point where the loss computed using the training data is at its lowest value while simultaneously the loss value with the same weights and bias values on of the validation data is at its lowest value. Continuing to train the network when the training loss decreases while the validation loss stays the same or even increases means we are overfitting the network. Overfitting a network means that the training data is modelled in too much detail and does not generalize well for other data. This is why the validation data set is utilized during the training in order to evaluate the model on a separate data set other than the training data. One way of detecting overfitting is to see if the training loss continues to decrease while the validation loss increases after some iterations. To find the best model for predicting new data, the training should stop before the network starts to overfit on the training data.

To avoid choosing a state of the model where the network has overfitted we can choose to save the model state each time the training reaches a new lowest validation loss. This means that we do not save the state when a network has overfit. The subsection below will describe the loss function and optimizer that we choose to use during training.

2.3.1 Loss function

In our models, we use the categorical cross-entropy loss because we are working with multi-class classification. Each model performs the task of classifying the most likely log event given some context. The categorical cross-entropy loss is a Cross-Entropy loss with a Softmax activation function. Usually, the softmax activation is applied to the output of the layers before the cross-entropy loss. The categorical cross-entropy loss will be used when we train the network to output a probability distribution for the classes for each input. The cross-entropy loss function is defined as in (2.13).

$$Loss(x) = -\sum_{i=1}^{K} y_i \cdot log f_i(x) \tag{2.13}$$

x is the sample (input for this layer), $\mathbf{y}$ is the one-hot encoded vector of the ground truth class so that y_i is either 1 at the position of the correct class and 0 at all other positions and K is the output size of this layer which corresponds to the total amount of classes. $f_i(x)$ is the corresponding output value (estimated probability that class i occurs). The minus sign ensures that the loss gets smaller when the estimated probability of the correct class increases. The softmax function is defined as in 2.14.

$$f_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.14}$$

In this function, the x_j are the output of the layers inferred by the net for each class in the output size, the softmax activation for a class x_j depends on all the output of the layers in input x.

2.3.2 Optimization algorithms

To update the weights and biases we utilize the gradients of the loss function with respect to each weight and bias. We can use the gradient of the loss function concerning each scalar entry to update the parameter values in gradient descent. The manner in which the weights and biases are updated are determined by the chosen optimization algorithm and the objective of the optimization algorithm is to minimize the loss. For this project, we have used Adaptive Moment Estimation (Adam) [16] but we will start with a brief introduction to mini-batch gradient descent to explain the optimizer.

Mini-batch gradient descent

Mini-batch descent is a gradient descent algorithm, the goal is to keep loss as small as possible. This algorithm splits all data into small batches to calculate loss and update parameters. *Batch* is the size of the samples processed before the model is updated (the size should be more than zero and less than the size of the entire training dataset).

A smaller batch size gives a learning process that quickly converges during training. Larger values of batch size provide a learning process that will slowly converge as the error gradient is accurately estimated. Usually the batch size is a power of two, that can fit the CPU/GPU memory. In the update function below, the Θ is the parameter and the η is the learning rate which is a hyper-parameter to control the size of the update steps along the gradient, the range of learning rate is between 0 to 1. n is the batch size, M is the number of batches where $M = \left\{1, \frac{N}{n}\right\}$, N is the number of the entire sample. The function will be updated the parameters at the end of each batch (denoted as e) and the function is:

$$\Theta_{e+1} = \Theta_e - \eta \frac{1}{n} \sum_{n \cdot M}^{n \cdot (M+1)} \bigtriangledown_{\Theta_e} Loss \tag{2.15}$$

Adaptive Moment Estimation (Adam)

Adaptive Moment Estimation (Adam) [16] is an algorithm that computes adaptive learning rates for each parameter. Equations 2.16 and 2.17 are the exponential moving average, they will update the value by each batch. Where m and v are moving averages, e and $e-1$ are the epoch steps, g is the gradient of the loss given the current batch, and β — new introduced hyper-parameters of the algorithm. The vectors of moving averages are initialized with zeros at the first iteration.

Equations 2.18 and 2.19 refer the bias correction, when e is bigger, the $1 - \beta_1^e$ and $1 - \beta_2^e$ will close to 1. Equation 2.20 will update the parameters and gives the output. The ϵ here is just a small value (default 10^{-7} in Keras [4]) to make sure the denominator of the equation is not zero. η is the learning rate.

$$m_e = \beta_1 m_{e-1} + (1 - \beta_1) g_e \tag{2.16}$$

$$v_e = \beta_2 v_{e-1} + (1 - \beta_2) g_e^2 \tag{2.17}$$

$$\hat{m_e} = \frac{m_e}{1 - \beta_1^e} \tag{2.18}$$

$$\hat{v_e} = \frac{v_e}{1 - \beta_2^e} \tag{2.19}$$

$$\Theta_{e+1} = \Theta_e - \frac{\eta}{\sqrt{\hat{v_e} + \varepsilon}} \hat{m_e} \tag{2.20}$$

When training the Transformer network with the Adam optimizer we noticed that the loss increased drastically after some epochs. Therefore we implemented the AMSGrad optimization algorithm [29] that claims to mitigate some issues that the Adam algorithm might have with convergence. The use of AMSGrad was also motivated by the fact that this was used in the paper where the Transformer was presented [31]. The difference between AMSGrad and Adam is that AMSGrad uses the maximum of past squared gradients to compute $\hat{v}_e$ instead of the exponential moving average. For all other networks, we have used the Adam optimizer instead of the AMSGrad optimizer because we did not notice any issues with convergence

for the other methods.

2.4 Components of the Neural Networks

Figures 2.2, 2.7 and 2.8 all depicted simple example structures of the neural network in the previous section. In short, every neural network includes at least one input and one output, with one or more layers in between. In this thesis we have used some specific layers that will now be presented here. The full architectures of the networks will be presented in section 3.3. We will begin by explaining the inputs and the outputs used in this project to provide a better understanding of the layers.

2.4.1 Inputs and outputs to the networks

To reiterate, the theme of our project is sequence modeling. The input to our network architectures are therefore sequences of integers corresponding to log lines that in turn explain events in the log. The meaning of the integers can be thought of as a class representing a certain event in the logs where the total amount of possible integers are the total amount of classes and this is a fixed amount corresponding to the possible events in a log. If we isolate one integer at one position in the sequence, this can be thought of as the current class, or the current log event in the specific time step. This integer can be represented as a one-hot encoded vector with the same dimension as there are classes (log events) with a 1 at the position of the integer value and 0 at all other positions. As a result, the inputs to the network are one-hot encoded vectors representing a class of log lines. We will call a class of log lines a *log key*.

The outputs of our networks are estimated probability distributions over all possible classes or events in a log. The way that the network will produce this output is explained in section 2.4.5. This ties back to the explanation of the loss function in section 2.3.1 where we explained that the input to the loss function is the result of a softmax operation.

2.4.2 Embedding layer

The embedding layer projects a one-hot encoded vector into a distributed representation of a different dimension. This requires an embedding matrix $W^{(emb)}$ initialized randomly that is then updated during the training of the network. Assume that we have a single log key integer that is one-hot (OH) encoded, $\mathbf{x}_i^{(OH)}$. The one-hot encoded log key vectors are of dimension K and the embedding matrix is of dimension $d_{(W^{(emb)})} = (K \times (\text{embedding size}))$. The one-hot encoded log key is multiplied with the embedding matrix resulting in a denser representation.

$$\mathbf{x}_i^{(emb)} = \mathbf{x}_i^{(OH)} W^{(emb)} \text{ for all } i \text{ in the input sequence} \tag{2.21}$$

The embedding size is an adjustable parameter in the embedding layer.

2.4.3 LSTM layer

One LSTM layer consists of a set of recurrently connected LSTM-cells. The LSTM layer has one changeable parameter which is the number of nodes in the hidden layers (dimensionality of the hidden layers) which is the number of cells in the forget gate layer. The output from the LSTM-layer at each time step i is the hidden state $\mathbf{h}_i$ and the cell state $\mathbf{c}_i$ as shown in 2.22. However, the output to be used as input into other layers is $\mathbf{h}_i$. For simplicity we combine all computations of the LSTM-cell explained in section 2.2.1 into a function $LSTM()$

$$\mathbf{h}_i, \mathbf{c}_i = LSTM(\mathbf{x}_i^{(emb)}, (\mathbf{h}_{i-1}, \mathbf{c}_{i-1})) \text{ for all } i \text{ in the input sequence} \tag{2.22}$$

2.4.4 Feed forward layer

In the feed forward fully connected (fc) layer all the neurons in the previous layer are connected to all the neurons in the next layer. A fully connected layer is the matrix multiplication between an input matrix and a weight matrix. This layer is normally placed before the output layer and often follows LSTM layers. The function $g()$ is one of the activation functions mentioned in section 2.1.1. The vector $\mathbf{x}_i^{(fc)}$ (2.23) refers to the output vector from the fully connected layer.

$$\mathbf{x}_i^{(fc)} = g(\mathbf{h}_i \mathbf{W}^{(fc)} + \mathbf{b}_i) \tag{2.23}$$

2.4.5 Output layer using a softmax activation function

$$\mathbf{O}_{i+1} = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \tag{2.24}$$

The output layer is responsible for producing the final result and in the sequence modelling tasks in this project we want to output an estimated probability distribution. The softmax function normalizes the input (vector of some real numbers) into a probability distribution consisting of K (all possible log key classes) estimated probabilities proportional to the exponential of the input size. The softmax function is presented in (2.25).

$$Softmax(f_{y_i}) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \tag{2.25}$$

The output of the softmax can be interpreted as a probability vector because the sum of each whole vector equate to 1. Thus, the output of the softmax layer is the estimated probability that the input belongs to a certain class out of all the classes which is the size of our vocabulary.

2.4.6 Attention layers

Recurrent neural networks with LSTM carry information forward through the cell state and the hidden state of the past time steps. However, when the length of the input sequences increase some information may be lost. To combat this *attention* can be incorporated in neural networks.

For example, attention can be implemented by including a so-called context vector [2] which saves information from each hidden state. There are, however, several different ways to implement attention mechanisms [20] but in this project we only use one attention-based network architecture, the Transformer [31]. In section 2.4.6.1 we will explain the mechanisms behind the attention mechanisms used in the Transformer which is called multi-head self-attention. The network architecture will be presented in section 3.3.4.

2.4.6.1 Multi-head self-attention layer

We will present the concept of multi-head self-attention by assuming we have some input sequence $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{ws}\}$ where the $\mathbf{x}_t$ at some time $t, t \in \{1, ..., T\}$ is a vector.

The self-attention is implemented by creating three vectors (often referred to as *query* ($\mathbf{q}_i$), *key* ($\mathbf{k}_i$) and *value* ($\mathbf{v}_i$)). To obtain these three vectors we utilize three matrices $\mathbf{W}^{(query)}$, $\mathbf{W}^{(key)}$, $\mathbf{W}^{(value)}$. Each matrix creates one of the three vectors indicated by its name. Assuming that we want to feed the vector embeddings $\mathbf{x}_i^{(emb)}$ of dimension $d_{(emb)}$ of a one-hot encoded log key $\mathbf{x}_i^{(OH)}$ the matrices are of dimension $d_{(emb)}$. The result are the vectors $\mathbf{q}_i$ (2.26), $\mathbf{k}_i$ (2.27) and $\mathbf{v}_i$ (2.28).

$$\mathbf{q}_i = \mathbf{x}_i^{(emb)} \mathbf{W}^{(query)} \tag{2.26}$$

$$\mathbf{k}_i = \mathbf{x}_i^{(emb)} \mathbf{W}^{(key)} \tag{2.27}$$

$$\mathbf{v}_i = \mathbf{x}_i^{(emb)} \mathbf{W}^{(value)} \tag{2.28}$$

The output $\mathbf{z}_i$ (2.29) of the self-attention layer is obtained by taking the sum of the product of the weights (2.30) by the value vectors $\mathbf{v}_j$.

$$\mathbf{z}_i = \sum_j w_{ij} \mathbf{v}_j \tag{2.29}$$

$$w_{ij} = softmax(\frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_{(emb)}}}) \tag{2.30}$$

Multi-head attention means that instead of outputting just one output vector $\mathbf{z}_i$ we output h different outputs $\mathbf{z}_i^k$, $k = 1, ..., h$ (2.31). The h parameter are called the number of *heads*. To obtain this we split the embedding vector $\mathbf{x}_i^{(emb)}$ into h chunks $\mathbf{x}_i^{(emb),k}$ of equal sizes of dimension $d_k, d_k < d_{(emb)}$. This requires h different matrices $\mathbf{W}^{(query),k}$, $\mathbf{W}^{(key),k}$, $\mathbf{W}^{(value),k}$ of dimension $d_{(k)} \times d_{(k)}$.

$$\mathbf{z}_i^k = \sum_j w_{ij}^k \mathbf{v}_j^k \tag{2.31}$$

$$w_{ij} = softmax(\frac{(\mathbf{q}_i^k)^T \mathbf{k}_j^k}{\sqrt{d_{(k)}}}) \tag{2.32}$$

Lastly, to obtain the final output each $\mathbf{z}_i^k$ (2.31) is concatenated and fed to a linear transformation to yield $\mathbf{z}_i$ of dimension $d_{(emb)}$.

3
Method

This chapter will describe the data and the methods used for anomaly detection. Starting with section 3.1 and section 3.2 we will explain the details regarding how we do data processing and create the data set that is to be used as input to the neural network architectures for the models. Section 3.3 will show the architectures of the neural network models. Then the following sections will present the methods of anomaly detection.

3.1 Data Description

As previously mentioned in Section 1.3 we have utilized two datasets in this project. To start with, we will present the Volvo GTT data set in Section 3.1.1 and this will be followed by a description of the HDFS data set in Section 3.1.2. Before we begin we will shortly explain the definitions of the labels in the data sets. The labels are indicative of whether a log sequence is an 'anomaly' or 'normal' which corresponds to the patterns of the sequences. The labels are on a log level where the log is a collection of temporally ordered log lines. A log file labeled as an 'anomaly' contains one or more sequential irregularities while a log file labeled as 'normal' is expected to have no irregularities.

Furthermore, we have distinguished between the data sets used in the process of training a neural network model and the data sets used in the process of evaluating the anomaly detection. For the Volvo GTT data set we have a separate labeled data set used for anomaly detection. For the case of the HDFS data set we need to create an anomaly detection data set which is a random sample of log sequences from the full data set.

3.1.1 Volvo Group Trucks Technology (Volvo GTT)

The Volvo GTT data set consists of a sample of software test output contained in text-files (`.txt`) during a time span of four months. Each of the files contains information from the test jobs and the text contained in the files can be split into lines, which we will refer to as *log lines*. The samples collected over four months contain no labels and it will be referred to as the *Model training data set*. This corresponds to the mixed data set in the HDFS data presented in section 3.1.2. That means that there is no knowledge about which log files contain one or more anomalies and which ones do not. To be able to evaluate the methods we received a small sample containing a total of 81 log files collected after the primary sample was collected and this data set will be referred to as the *Anomaly detection data set*. The Anomaly detection data set has not been used for training any of the models.

	Model training data set	**Anomaly detection data set**
Total number of log files	34 083	81
Total number of log lines	13 436 501	59 601
Number of log files labeled as anomalous	Unknown	8

Table 3.1: Description of the data sets where the Model training data set are the amount of log files used when training the neural network models and the Anomaly detection data set is the data set used when detecting anomalies for the Volvo GTT data set.

In detail, each log file consists of several executed tests. The tests are distinguishable inside the log file, with a command presenting the test being executed and some subsequent responses to the test (whether it was successful or if the response was abnormal). The data contains human-readable text with time stamps and some semi-structured text such as Extensible Markup Language (XML).

During the course of the project we found that splitting each log file on the distinguished tests inside a log file before anomaly detection improved the performance of the anomaly detection. However, it was better to train the neural networks on the data without using this split which is why we did this afterwards. We assume that this is due to the fact that the different tests are not always executed in a certain order and this is the format of the data we have used to produce the results.

3.1.2 Hadoop Distributed File System (HDFS)

The Hadoop distributed file system log (HDFS) dataset [38] used in this project contains 575 061 log sequences called blocks that are labelled either as 'normal' or as 'anomaly'. It is a publicly available dataset and was used in this project to have as comparison to the Volvo GTT dataset which only contains a small proportion of labelled data.

The dataset was retrieved from Loghub Zenodo [38]. In its raw form it only contains the log output as unstructured text produced by concurrent processes and the labels are in a separate table containing labels for each block-id. In order to use this data as separate log sequences we utilized the block-id:s of each line to group the log lines together that belonged to one block id. This process can be found in the preprocessing implementation details provided by LogPAI [14]. The result is a dataset containing a log sequence for each block-id representing a single process.

To obtain separate datasets used for training the networks and performing anomaly detection we split the entire dataset randomly where 80% of the data was chosen to be the Model training data and 20% to be the anomaly detection data. Figure 3.3 shows the workflow of splitting dataset.

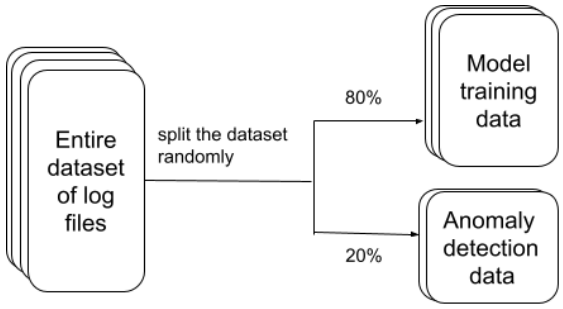

Figure 3.1: Workflow of how the HDFS data set is split into a model training data set and an anomaly detection data set.

In addition, we create two types of training datasets: Model training (mixed dataset) includes block-id:s with 'anomaly' label. Then we remove all the block-id:s with 'anomaly' label from the Model training (mixed dataset) to create Model training (normal dataset). Table 3.2 contains an overview of the datasets.

	Model training (mixed dataset)	**Model training (normal dataset)**	**Anomaly detection dataset**
Number of block-id:s	460 048	446 541	115 013
Number of block-id:s labeled as anomalous	13 507 (3%)	0	3 331 (2,9%)

Table 3.2: Description of the mixed data set and the data set containing only normal log files for the HDFS data set.

3.2 Data Preprocessing

To be able to use the log file data in the sequence learning models it needs to be converted to numerical values. In natural language processing [10] this is commonly done by going through the entire data set consisting of text and extracting all unique words. After this a dictionary can be created with all unique words, and in this dictionary an integer is mapped to each of these unique words. This means that each particular word can be identified by an integer. The process of mapping text to an integer and creating a dictionary is called tokenization [10].

We have used tokenization in this project but not on a word level, but instead on a log line level. However, log lines often consist of various variable components such as time stamps and parameter values. The variable components can best be explained by a simple example print statement in a program such as `print('The value of variable x is %d')`. When running the program we then get a statement and let us say that this is `The value of variable x is 0`. The *static* component of this print statement is `The value of variable x is *` while the *variable* component is the numeric value which is 0.

The variable components, especially in the case of time stamps, introduce the issue of a very large dictionary if we were to tokenize these directly. In addition, if the time stamps are present each newly created log line would be a unique word in the dictionary unless the exact same log statement was created at the exact same time. There is little meaning in the patterns produced if every single log line is its own unique entry in the dictionary. Because of this we need to extract the variable components from the log lines, we will call this cleaning the log lines.

Continuing on the topic of cleaning the log lines, we used regular expressions (regex) to remove variable components that follow a certain pattern. One example of a pattern is the pattern of a timestamp, which could be `YEAR:MONTH:DAY HOUR:MINUTE:SECOND`. Regex can remove these types of patterns and we have used the following patterns to clean the data consisting of log lines:

- Numerical values

- Non-alphanumeric characters
- Replace empty lines with a string (`empty_line`)

After cleaning the log lines, we create a dictionary mapping each unique clean log line to an integer which is refered to as the *log key*. The dictionary can then be used to produce integer sequences where each integer in the sequence correspond to a cleaned log line.

An example of the data cleaning is found in figure 3.2, with a log line from the HDFS dataset.

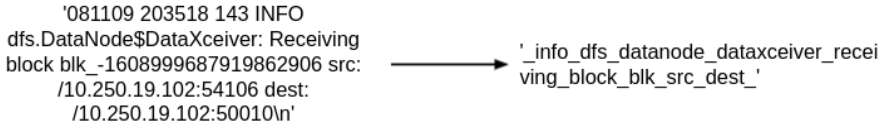

Figure 3.2: Example of preprocessing done with regex of a log line in the HDFS data set.

The order of converting the raw log file data to an integer sequence is therefore as described below:

- Split each log file into a sequence of log lines
- Clean each log line with the stated regex methods to obtain the static version of a log line
- Convert the static versions of the log lines of the log into integers using the dictionary. This creates a sequence of integers also called a log sequence.

The result of this preprocessing is a sequence of integers which represents sequences of temporally ordered events in the log. The entire workflow has been visualized in figure 3.3.

The resulting log integer sequences are of variable size, and can be long. This introduces an issue regarding long-term dependencies with sequence modelling. Because of this we used a so-called *sliding window* technique to decrease the length of the sequences to perform pattern recognition on. This method will be explained in section 3.2.1.

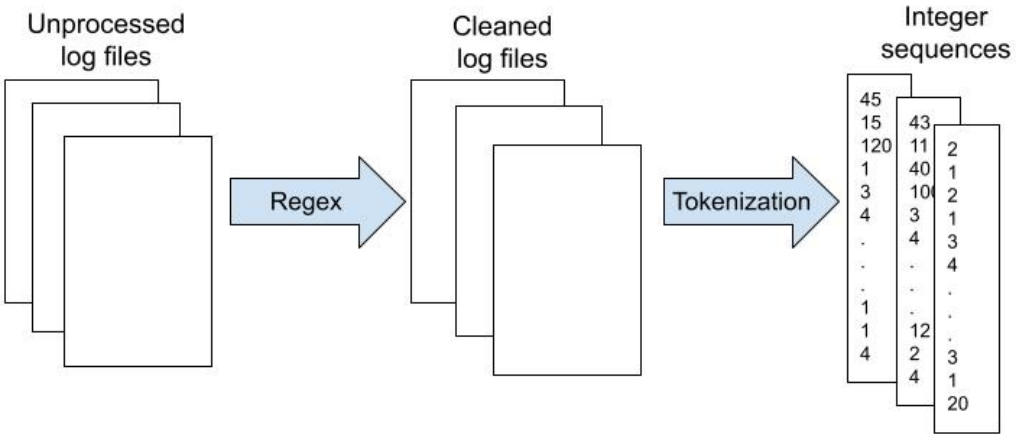

Figure 3.3: The data pre-processing flow going from a completely unprocessed set of log files to the integer sequences representing the log files.

3.2.1 Sliding window technique

The idea behind the sliding window technique is to traverse through a log sequence and divide the sequence into fixed size windows while still preserving the temporal order of the integers.

Sliding windows require a window size and a step size. The window size determines how long a window should be while the step size determines the step taken while creating one window to the next. In this project we have used a window size of 10 and a step size of 1 for all models, which is what we found was the best for the Volvo GTT data. This creates overlapping windows from the log integer sequence.

The sliding window technique is described in (3.1) as the creation of sliding windows from an example integer sequence. In this example, the sequence length is 5, the window size is 3 and the step size is 1.

$$\begin{bmatrix} 2 \\ 1 \\ 3 \\ 20 \\ 5 \end{bmatrix} \rightarrow \begin{matrix} [2 & 1 & 3] \\ [1 & 3 & 20] \\ [3 & 20 & 5] \end{matrix} \tag{3.1}$$

3.2.2 Vector representation of log keys

The inputs into each of the networks that will be presented in section 3.3 are vector representations of the log keys. As explained in previous sections 3.2 and 3.2.1 we create sequences of log keys by assigning an integer to each unique cleaned log line. The cleaned log lines and their corresponding integer then composes the dictionary of all log keys. This process enables the log keys to be represented in vector format by one-hot encoding the log keys.

A one-hot (OH) encoded log key is a vector the same size as the size of the dictionary, or the vocabulary size. The vectors has zeros in all positions except for the position indicated by the log key integer. Assume that the log key integer is 2 and the size of the vocabulary is 4. A one-hot encoded representation of this log key is then as in (3.2).

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{3.2}$$

3.3 Network Architectures and Methods for Prediction

In this section we will present the architectures of the networks. So if we think of the layers presented in section 2.4 as the building blocks of the neural networks, we now want to present the entire structure. Furthermore, we want to explain how these models learn the patterns of the sequences and how they can be used for anomaly detection.

We have utilized four different architectures to compare anomaly detection performances. The four different architectures can be divided into two main approaches. The first approach is to teach the model to predict a subsequent log key given some previous context window from the log sequence. The second approach is to try to reconstruct the context window.

The first approach is similar to the approach used DeepLog [7]. A recurrent neural network with LSTM-cells presented in 3.3.1 is used with the intention of creating a predictor for the subsequent log key given a window of past log keys of a fixed size. We also tried an extension to this, a bidirectional network presented in 3.3.2, which draws information from both past and future information from the sequence. Moreover, two sequence-to-sequence models were evaluated which include an LSTM-based model presented in 3.3.3 and a Transformer presented in 3.3.4.

Notation

$$ws := \text{window size} \tag{3.3}$$
$$T := \text{indices of time step in the window of size } ws, t \in T = \{1, ..., ws\} \tag{3.4}$$
$$\ell_t := \text{log key at time step t} \tag{3.5}$$
$$\ell_0 := \text{special token log key 'sos'} \tag{3.6}$$
$$\mathbf{x}_t := \text{one-hot encoding of one log key } \ell_t \text{ at time step t} \tag{3.7}$$
$$\mathbf{O}_t := \text{output from the model corresponding to time step t} \tag{3.8}$$
$$\hat{\ell}_t := argmax(\mathbf{O}_t), \text{ predicted log key at time step t} \tag{3.9}$$

3.3.1 Unidirectional Long Short Term Memory Network (uni-LSTM)

In the unidirectional LSTM (uni-LSTM) network the objective of the training of the neural network is to learn to predict a log key integer given log keys from past time steps of a fixed window size. In Figure 3.4 we have a scheme of the architecture of this network.

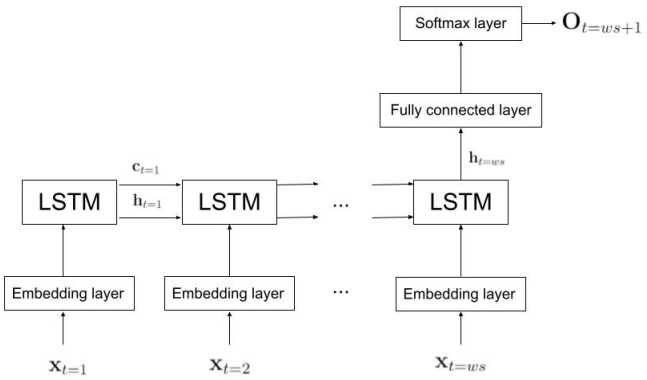

Figure 3.4: Visualization of the unidirectional recurrent neural network with LSTM-cells unfolded in time that predicts the next coming log key given a window of window size (ws). The last layer, the softmax layer, generates an estimated probability distribution over all possible log keys.

The input into the network presented in Figure 3.4 are one-hot encoded log keys which represent the log keys at a certain time step in the window. The one-hot encoded log keys are fed to an embedding layer (performing the operation presented in Section 2.21). The embedding layer projects the input to another dimension and makes the vector a distributed vector instead of a one-hot encoded vector. The distributed vector is fed to an LSTM-cell and the cell state and hidden state is carried forward over time. The last output from the LSTM-layer is fed to a fully connected layer to project the vector to a dimension the same size as the size of the dictionary. The output from the fully connected layer is then fed to a softmax layer that computes the estimated probability distribution over all log keys given information of a fixed length (the window size).

The predictions are received by finding the log key with the highest estimated probability. The output is $\hat{\ell}_{t=ws+1} = argmax(\mathbf{O}_{t=ws+1})$ if we wanted to output the most likely log key. For anomaly detection we use the *top-n* method [7] that will be presented in Section 3.4.

3.3.2 Bidirectional Long Short Term Memory Network (bi-LSTM)

In the bidirectional LSTM (bi-LSTM) network there are basically two independent LSTM networks coming from different directions in the log sequence. The network is therefore constructed to predict a log key given the past and the future. The architecture can be seen in figure 3.5.

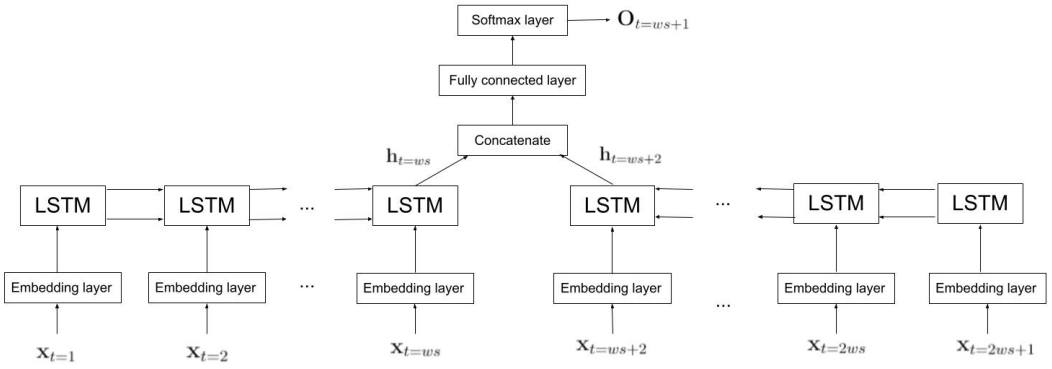

Figure 3.5: Visualization of the bidirectional recurrent neural network with LSTM-cells unfolded in time that predicts a log key given a window of window size (ws) taking information both from the past and the future. The last layer, the softmax layer, generates an estimated probability distribution over all possible log keys.

The process in the bi-LSTM is the same as for the uni-LSTM, presented in section 3.3.1 with the exception that we have two LSTM layers coming from different directions. One of the LSTM layers take a context of a certain window size from the past and the other takes a context of a certain window size from the future. The estimated probability distribution coming from the softmax layer is therefore instead the probabilities of the log key given the past and the future.

The same method for prediction is used for the bi-LSTM model as for the uni-LSTM model with the exception that we now want to predict $\hat{\ell}_{t=ws+1} = argmax(\mathbf{O}_{t=ws+1})$ which is a result of the concatenation of the vectors $\mathbf{h}_{t=ws}$ and $\mathbf{h}_{t=ws+2}$ fed through a fully connected layer and a softmax layer. The anomaly detection method is the same as for the uni-LSTM model, the top-n method, presented in section 3.4.

3.3.3 Sequence-to-sequence LSTM Architecture

In a sequence-to-sequence LSTM architecture both the input and the output consists of a sequence of a certain window size. Similar architectures are used in for example machine translation [30] mentioned in section 1.2. The sequence-to-sequence architecture can be divided into two separate networks: an encoder and a decoder, as presented in figure 3.6.

The arrows between the encoder and the decoder represent the hidden state ($\mathbf{h}_{ws}$) and the cell state ($\mathbf{c}_{ws}$) (Figure 2.9) after the last input into the encoder. The cell state and the hidden state are used as the initial states in the decoder-network. The decoder is used to reconstruct the input based on these states.

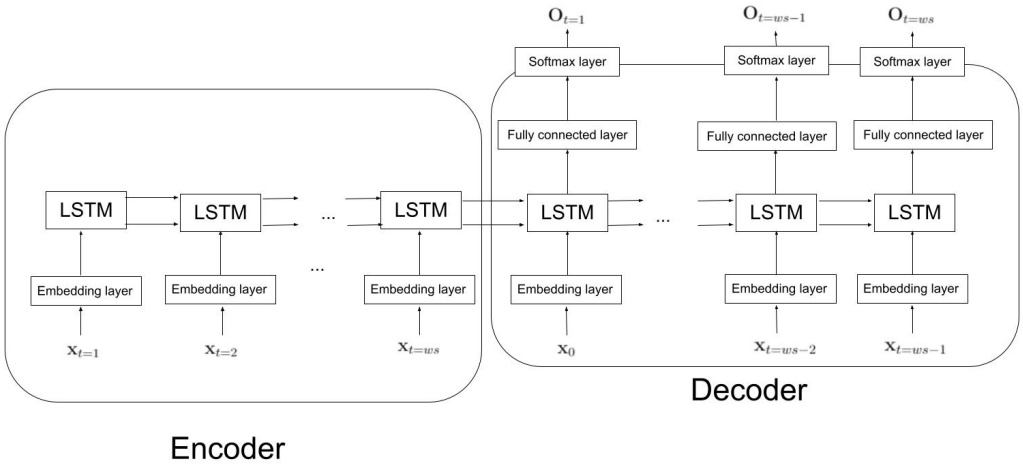

Figure 3.6: Visualization of the LSTM sequence-to-sequence autoencoder architecture where the input to the decoder is the input used during training.

In the training stage of the model, the input window is fed to the encoder. It is this input that is visualized in Figure 3.6. The sequence fed to the decoder contains the vector representation $\mathbf{x}_0$ of a special token ℓ_0 (3.6) representing a *start of sequence* token and has one less element than the encoder input at the end. This enables the network to learn to predict the coming log keys. This means that the input into the decoder is of the same format as presented in 3.11. This is called *teacher forcing* [30].

$$\mathbf{X}^{(encoder)} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{ws}\} \tag{3.10}$$

$$\mathbf{X}^{(decoder)} = \{\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{ws-1}\} \tag{3.11}$$

During prediction when we use the model for anomaly detection there is no teacher forcing, instead the predicted log key from the previous time step is used as input to the decoder. To predict the first log key ℓ_1 we utilize the special token ℓ_0. This predicted log key is then fed to the decoder in the next step and this procedure keeps going until a sequence of the window size has been created. The algorithm is explained in detail in Algorithm 1.

Algorithm 1: Prediction algorithm for the LSTM sequence-to-sequence model

Result: Sequence of predicted reconstructed log keys $\{\hat{\ell}_1, \hat{\ell}_2, ..., \hat{\ell}_{ws}\}$
Feed $\mathbf{X}^{(encoder)}=\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{ws}\}$ to the encoder to obtain hidden state $\mathbf{h}_{ws}$ and cell state $\mathbf{c}_{ws}$;
Feed vector representation of special token ℓ_0 to the decoder and obtain $\mathbf{O}_1$;
Save $\hat{\ell}_1 = argmax(\mathbf{O}_1)$ as the predicted log key for $t = 1$;
for *t=2,...,ws* **do**
    Feed vector representation of $\hat{\ell}_{t-1}$ to the decoder and obtain $\mathbf{O}_t$;
    Save $\hat{\ell}_t$=$argmax(\mathbf{O}_t)$ as the reconstructed log key at time t;
end

3.3.4 Sequence-to-sequence Transformer Architecture

This architecture is the same architecture as presented in [31]. We will present it here to explain how we have used the network. The Transformer performs the same task as the LSTM sequence-to-sequence network presented in the previous section (section 3.3.3). The main difference is that no recurrent mechanism is used for this network. Instead all of the layers are feed-forward layers and self-attention layers. To keep track of the positions of the log keys in the input sequence two operations are performed: a positional embedding and a sequence embedding.

For an explanation of what happens in the embedding layers we refer to section 2.4.2. The task of the sequence embedding is the same as in the LSTM networks. In the case of the Transformer, the positional embedding serves to project the one-hot encoded *position* of a log key at a temporal position in the sequence into a lower dimension. This is done in order to preserve information about the order of the log keys in the sequence.

The multi-head attention, as explained in section 2.4.6.1 is positioned at three places in the model and the output from the attention layer is added and normalized with the input into the attention layer. After addition and normalization the vectors are fed to a feed forward layer as presented in Section 2.4.4 with a ReLU activation function which was presented in Section 2.1.1.

The encoder and the decoder both contain a block of four layers that are presented in figure 3.7. In the following explanations we will refer to this block as the Transformer block.

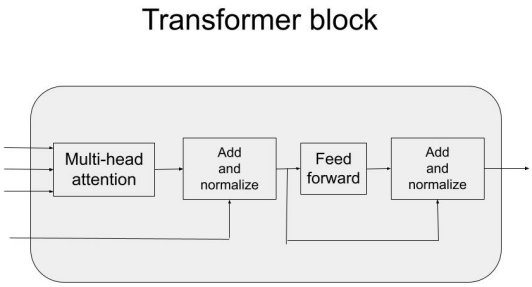

Figure 3.7: Visualization of the Transformer block used to build the Transformer.

The entire network of the Transformer is presented in figure 3.8 and we will now shortly explain the workings of the network, starting with the encoder. When input is fed to the encoder this means that the input positions and the input sequences are fed to an embedding layer and then combined by vector addition. The resulting vector is copied 4 times and fed to a Transformer block. The encoder computations are repeated N times where the output of the encoder is fed back to the encoder. The last output is then fed to the transformer block in the decoder along with the vectors created by the previous operations in the decoder (as can be seen in figure 3.8). The decoder operations are also repeated an N amount of times in the same manner as for the encoder. Finally, before the softmax layer which computes the estimated output probabilities a linear transformation $\mathbf{x}_i^{out} = \mathbf{x}_i^{in}\mathbf{W}^l + \mathbf{b}$ is performed on the vectors fed to the layer.

During training the network uses teacher forcing in the same manner as explained in section 3.3.3. The difference in the Transformer is that no recurrent mechanism is utilized. Instead, there is a masked multi-head attention present in the decoder. The masked multi-head attention prevents the network from accessing future log keys in the window. Assume that we have some window of log keys from a log sequence $\{55, 10, 1, 43, 1, 2, 3, 9, 75, 30\}$ that we want to reconstruct. The input into the decoder is then $\{'sos - token', 55, 10, 1, 43, 1, 2, 3, 9, 75\}$. At time step 0 the decoder receives the $'sos - token'$, at time step 1 the decoder receives the $\{'sos - token', 55\}$ and this continues until it has received the entire decoder input sequence. After the network has been trained and the model is used for prediction we follow the algorithm presented in Algorithm 2.

Algorithm 2: Prediction algorithm for the Transformer sequence-to-sequence model

Result: Sequence of reconstructed log keys $\{\hat{\ell}_1, \hat{\ell}_2, ..., \hat{\ell}_{ws}\}$
Feed $\mathbf{X}^{(encoder)}$=$\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{ws}\}$ to the encoder to obtain the queries $\mathbf{q}$ and keys $\mathbf{k}$;
Feed ℓ_0 to the decoder and obtain $\hat{\ell}_1 = argmax(\mathbf{O}_1)$;
Save $\hat{\ell}_1$ as the predicted log key for $t = 1$;
for *t=2,...,ws* **do**
    Feed $\{\ell_0, \hat{\ell}_1, ..., \hat{\ell}_{t-1}\}$ to the decoder and obtain $\hat{\ell}_t = argmax(\mathbf{O}_t)$;
    Save $\hat{\ell}_t$ as the reconstructed log key at time t;
end

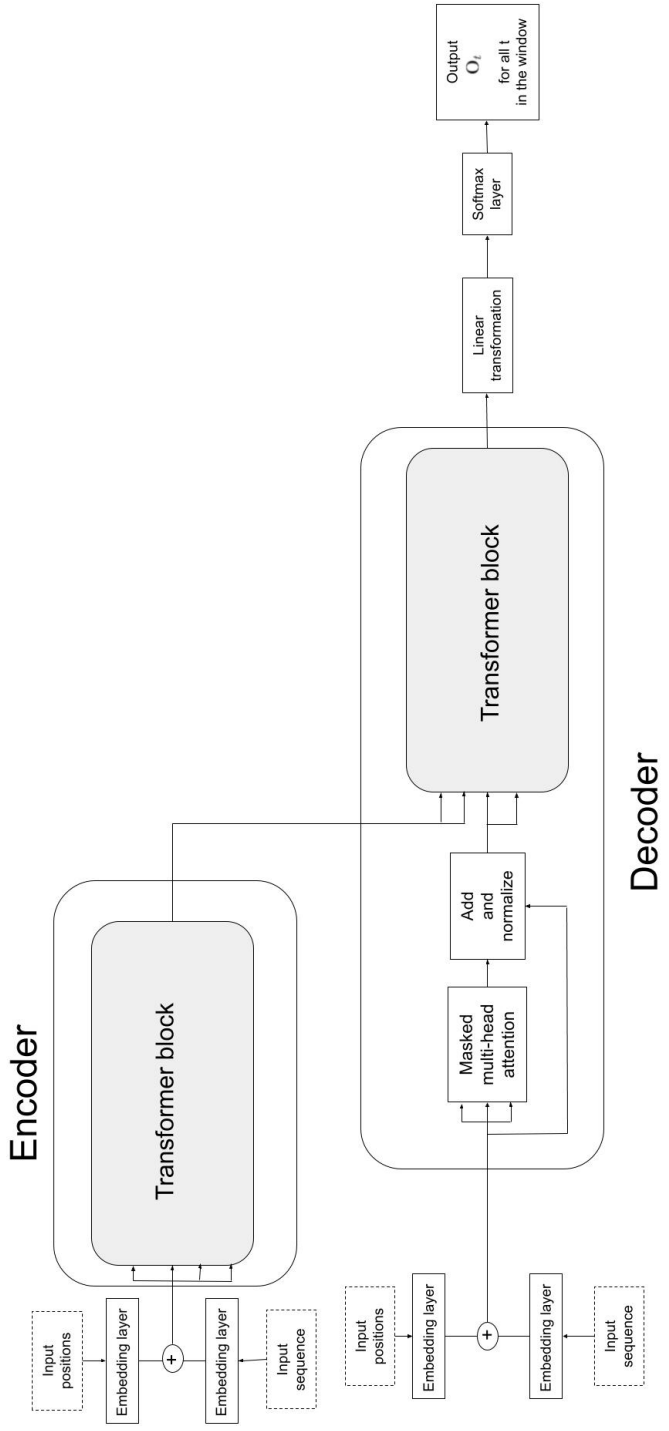

Figure 3.8: Visualization of the Transformer network.

3.4 Detecting irregularities from the model predictions

In the uni- and bi-directional LSTM networks presented in previous sections 3.3.1 and 3.3.2 we utilized a method which we will call the *top-n* method to detect irregularities. The top-n method of anomaly detection was presented in DeepLog [7]. The method consist of sorting the softmax output containing an estimated probability distribution from the model in descending order. Then if the true subsequent log key is not among the top n most likely subsequent log keys we have encountered an irregularity. The irregularities are used to detect anomalous log files. The entire method of detecting an anomalous log file will be explained in section 3.5.

The disadvantage of this method is the fact that the parameter n needs to be defined and considering there is no prior knowledge of what could be considered an anomaly in the Volvo GTT data set we needed to find this threshold n with trial-and-error. In the results presented in chapter 4 we have just shown the results for the best threshold n. This motivates the use of the sequence-to-sequence architectures, where we aim to get rid of the thresholds. In the sequence-to-sequence architectures, we are simply assumed to have found an irregularity if the window was not reconstructed correctly.

3.5 Detecting Anomalous Log Files from Model Predictions

During the training of the neural network models used in this project the objective is to create a model that either predicts the target log key correctly as in the uni- and bi-LSTM models presented in sections 3.3.1 and 3.3.2 or reconstructs the input sequence correctly as the sequence-to-sequence models presented in sections 3.3.3 and 3.3.4. The assumption that will be tested is that log lines corresponding to an anomaly is rare and following this the neural network models will not be able to learn these patterns.

This is on the level of a window (section 3.2.1) of log keys. We want to point to the *log files* that contain an anomaly. Because of this we now define what we will consider to be an anomalous log file and the background to the results in section 3.5.1. We then move on to explaining the evaluation metrics in section 3.5.1.2.

3.5.1 Log File Anomaly Definition and Metrics

3.5.1.1 Definition of an Anomalous Log File

The anomaly detection takes place on a file level and therefore a log file is defined as anomalous if it contains one or more wrong predictions. This is further motivated by that the data sets utilized in this project contain labels on a log sequence level. For the HDFS data set this means that each block-id has a corresponding label and in the Volvo GTT data set each log file in the evaluation data set has a corresponding label.

During anomaly detection for one log file we therefore follow the procedure below:

- Read the log file and preprocess the log file so that we receive the corresponding log integer sequence
- Convert the log integer sequence into windows of equal window size
- Perform the prediction algorithm from the respective methods
- if $\frac{\#\text{correct predictions}}{\#\text{all predictions}} < 1 \rightarrow$ log file is predicted anomalous

The reason why this definition was used is that in the Volvo GTT data set a log file/log sequence could be labeled as an anomaly but this does not necessarily mean that each line in that log sequence is anomalous. Because the log file data contains the result of software testing this means that some parts of the code could be working correctly and be non-anomalous, while some other part of the code is anomalous. It is also difficult to label the logs on a log line level as several log lines could express the anomaly.

3.5.1.2 Evaluation metrics

The anomaly detection method is a binary classification task on a log file level with the two classes anomalous (*positive*) and non-anomalous (*negative*). If an anomaly has been detected this is considered a positive prediction (3.12), and this happens when not all log lines are predicted correctly, that is if $\frac{\#\text{correct predictions}}{\#\text{all predictions}} < 1$. The method has classified a log file as non-anomalous if there are no wrongly predicted log lines, and this is considered a negative prediction (3.13).

$$\textbf{Positive prediction} := \text{log file is classified as anomalous} \tag{3.12}$$

$$\textbf{Negative prediction} := \text{log file is classified as non-anomalous} \tag{3.13}$$

To evaluate the anomaly detector the occurrences of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) were measured. The meaning of a true prediction is that the label on the log file corresponds to the prediction, while a false prediction is one where the label does not correspond to the prediction. Table 3.3 depicts all possible scenarios.

		Ground truth	
		Anomalous	*Non-anomalous*
Prediction	*Anomalous*	True positive (TP)	False positive (FP)
	Non-Anomalous	False negative (FN)	True negative (TN)

Table 3.3: Table of metrics.

From these scenarious the evaluation metrics presented in (3.14), (3.15), (3.16) and (3.17) will be used.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.14}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.15}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.16}$$

$$F1 = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \tag{3.17}$$

The accuracy (3.14) measures how many correct predictions there are in relation to the total amount of predictions. However, the accuracy can be high if the model only detects the true negatives which is not a useful model. Therefore the precision (3.15) which shows the rate of true positives in comparison to all positive predictions. The recall (3.16) shows the rate of true positives to all predictions of a log file labelled as an anomaly. The F1-score (3.17) is a combined metric for the precision and recall.

3.6 Implementation

The many-to-one recurrent LSTM models were implemented in Keras [4] with the TensorFlow framework. These models were trained on a CPU.

The autoencoder models were implemented with Pytorch [27] and trained using a Nvidia GTX 1080Ti graphics processing unit (GPU)

4
Results

The following sections will present the results for the Volvo GTT (section 4.2) dataset and the HDFS (section 4.3) dataset. The results are presented using training datasets of different sizes. The sections will start with details regarding the size of the dictionary and the amount of windows in the data sets using a window size of 10 and then move on to the anomaly detection results for the different methods. Finally we will compare the results of the best performing models for each data set.

All of the models have been trained with 10 epochs. The training converged quickly for the models and we did not detect any overfitting with the Adam optimizer and an initial learning rate of 10^{-4} for the uni-LSTM and bi-LSTM models. The parameter values used in the Adam optimizer was $\beta_1 = 0.9$ and $\beta_2 = 0.999$ which are the default values for the Adam optimizer in Keras [4].

Moving on to the Transformer, using the entire data set and the Adam optimizer with a learning rate of 10^{-4} introduced a problem with a drastically increasing loss value after a certain amount of epochs. Because of this we used AMSGrad [29] for the training of the Transformer. The Transformer models have been trained for 10 epochs using the Adam optimizer [16] with $\beta_1 = 0.9, \beta_2 = 0.98$ and $\epsilon = 10^{-9}$ with AMSgrad [29] which solved this issue. The parameters are the same ones used in [31].

For all models we have used mini-batch training which means that during one epoch the training samples are sent to the model in batches. For this end we tried batch sizes between $64, 128, 256$ and 512. We did not notice any significant difference when training the models and used 128 for the Volvo GTT data set and 64 for the HDFS data set as batch size in the results presented in section 4. The batch size determines how many samples of windows as presented in section 3.2.1 the model should be exposed to before updating the weights.

4.1 Data Overview

In this section we present the results from the pre-processing of the log files. All of the data sets were divided into windows of window size 10 (windows of window sizes 5 for the bi-LSTM models which results in a context length of 10) and the amount of windows present the total amount of windows on which the models were trained on. The Volvo GTT data set has a dictionary size of 533 and is presented in table 4.1 and the HDFS data set is presented in table 4.2.

Proportion of the data set	1%	5%	10%	100%
Number of windows (window size 10)	91 528	457 643	915 286	9 152 860

Table 4.1: Description of the amount of windows used for the Volvo GTT data set.

The HDFS data set is divided according to two different methods. In the first data set corresponding to the (mixed data) data sets in table 4.2 log sequences labeled as anomalies and normal are mixed. The data set corresponding to the (normal data) data sets in the table contain only log sequences that are confirmed normal. The Model training (mixed dataset) has a dictionary size of 97 and the Model training (normal dataset) has a dictionary size of 55.

	Model training (mixed dataset)				**Model training (normal dataset)**			
Propottion of the data set	1%	5%	10%	100%	1%	5%	10%	100%
Number of windows (window size 10)	42415	212074	424148	4241476	43767	218838	437676	4376760

Table 4.2: Description of the amount of windows used for the mixed data set (mixed data) and the data set containing only normal log files (normal data) respectively for the HDFS data set.

4.2 Volvo GTT

Architecture	Layers	Hidden size	Heads	Batch size	Epochs
uni-LSTM	1	128	-	128	10
bi-LSTM	2	128	-	128	10
LSTM seq2seq	3	128	-	128	10
Transformer seq2seq	6	256	8	128	10

Table 4.3: Summary of all the model parameters used for producing the results when training models with data provided by Volvo GTT.

For the bi- and uni-LSTM models on the Volvo GTT data we tried hidden sizes $64, 128$ and 256. The hidden sizes here refer to the embedding size, the size of the hidden state vector in the LSTM cells and the size of the output vector from the fully connected layer (Section 3.3.1 and 3.3.2). The models performed the best with a hidden size of 128 so this is what we have used in the results.

For the LSTM sequence-to-sequence model we found that more than 1 LSTM layer performed the best with the best amount of layers being 3. The hidden size of 128 also performed the best here, although this was still an unusable model for anomaly detection. For the Transformer, we used a hidden size of 256 with 8 heads.

4.2.1 Uni-LSTM model

The uni-LSTM models use the same method as the LSTM sequential anomaly detector in DeepLog [7]. The results are for the best thresholds which have been included in Table 4.4.

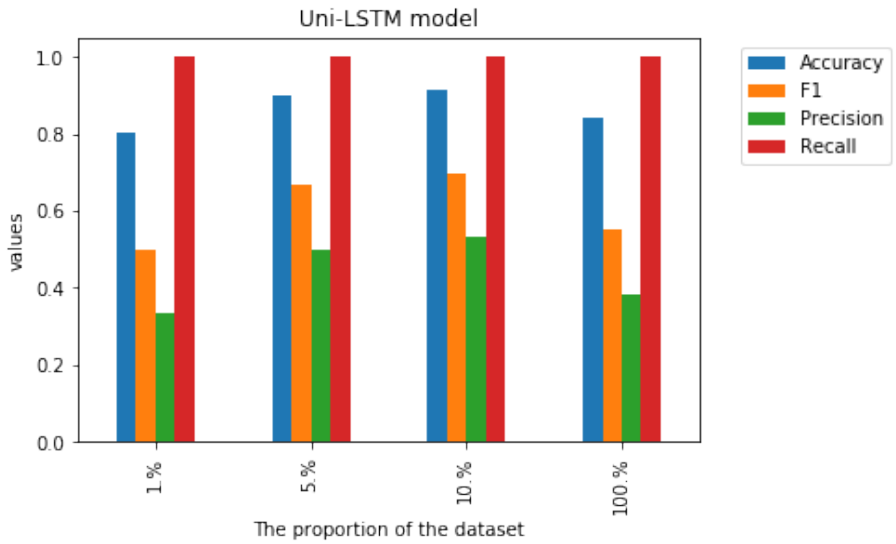

Figure 4.1: Results of the uni-LSTM based anomaly detectors for various proportions of the training data on the Volvo GTT data.

The uni-LSTM manages to catch all the log files labeled as anomalies for the different proportions of the data set. The best performing model is therefore the one that classifies the least false positives. This is the model trained with 10% of the data set. The TP, FP, FN and TN scores can be found in Table 4.4. Notably in all cases using the uni-LSTM we did not observe any false negatives (FN) which means that the model successfully catches the true anomalies.

Proportion of the dataset	Threshold	TP	FP	FN	TN
1%	20	8	16	0	57
5%	9	8	8	0	65
10%	9	8	7	0	66
100%	5	8	13	0	60

Table 4.4: True positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) for the uni-LSTM models with varying proportions of the data set anomaly detection results on the Volvo GTT data.

4.2.2 Bi-LSTM model

In contrast to the uni-LSTM models, the bi-LSTM models produce false negatives (FN) when using 10% and 100% of the data set. The model trained using only 1% of the data produces the overall best result indicated by the F1-score.

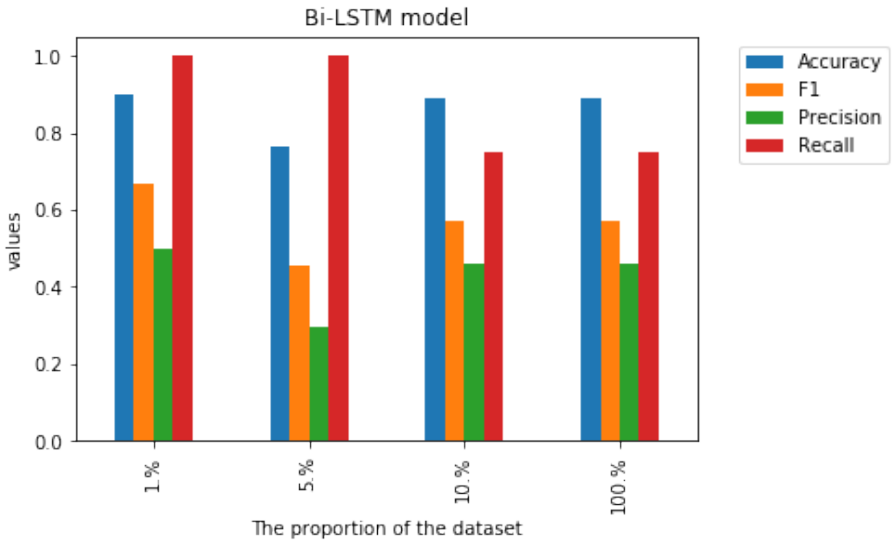

Figure 4.2: Results of the bi-LSTM based anomaly detectors for various proportions of the training data on the Volvo GTT data.

In Table 4.5 we can see the details and the thresholds used for producing these results. For this model there are cases when the model does not catch all anomalies as opposed to the uni-LSTM model. This means that if this model was to be used for anomaly detection important information would be missed meaning that this is not a preferable anomaly detection method to the uni-LSTM model.

Proportion of the dataset	Threshold	TP	FP	FN	TN
1%	8	8	8	0	65
5%	3	8	19	0	54
10%	5	6	7	2	66
100%	5	6	7	2	66

Table 4.5: True positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) for the bi-LSTM models with varying proportions of the data set anomaly detection results on the Volvo GTT data.

4.2.3 LSTM sequence-to-sequence model

When evaluating the anomaly detection performance of the LSTM sequence-to-sequence model the result was that no log files were ever predicted correctly. In order for a log file to be considered non-anomalous all of the windows of log keys in the log file should be predicted correctly, and this never happened in these results. The minimum accuracy for one log file was 0.451 and the maximum accuracy was 0.555 which means that it detects anomalies in all of the log files presented to it. This resulted in the amount of true positives being 8 and the amount of false positives being 73.

4.2.4 Transformer sequence-to-sequence model

While creating the anomaly detector and choosing hyperparameters for the Transformer we noticed that less data produced better results which can be seen in Figure 4.3 and Table 4.6.

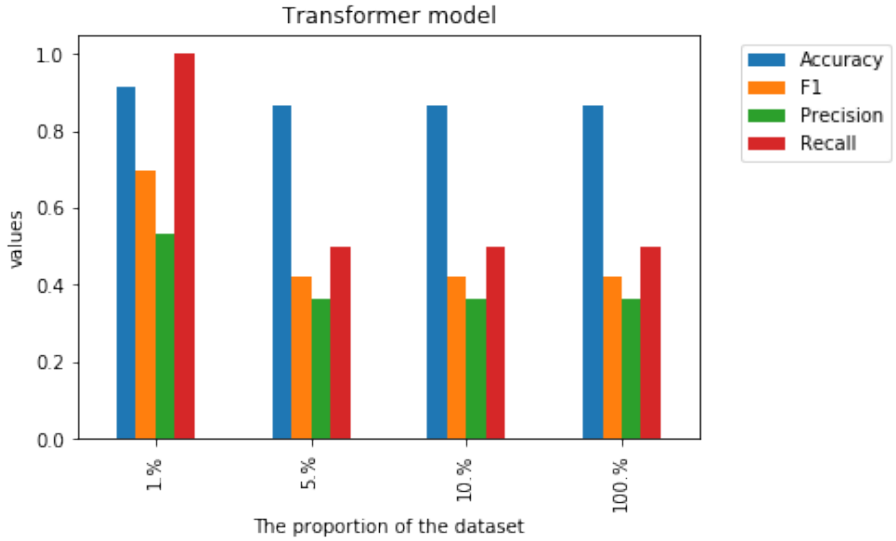

Figure 4.3: Results of the Transformer based anomaly detectors for various proportions of the training data on the Volvo GTT data.

The amount of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) are presented in table 4.6 for reference. The model using 1% of the data performed the best and when we increased the size of the data set the model performs worse. One interesting observation is that the false positives (FP) stays the same at a value of 7. This was also the lowest observed value in the uni- and bi-LSTM models. Nevertheless, a model missing anomalies is not a useful model.

Proportion of the dataset	TP	FP	FN	TN
1%	8	7	0	66
5%	4	7	4	66
10%	4	7	4	66
100%	2	7	6	66

Table 4.6: True positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) for the Transformer mosel with varying proportions of the data set anomaly detection results on the Volvo GTT data.

4.3 HDFS

For the Uni- and Bi-LSTM models on the HDFS data we have tried the same hidden sizes 128 as Volvo GTT data first. However, the model performed not so well with hidden sizes 128. Then we changed the hidden size to 64 and the model performed better, so hidden sizes 64 is what we have used in the results.

For the LSTM sequence-to-sequence model, the amount of layers where the model performed the best we found was 3 with a hidden size of 64, although this was an unusable model for anomaly detection. For the Transformer model, we used a hidden size of 64 with 4 heads.

Table 4.7 presents the summary of the all the model parameters used for producing the results.

Architecture	**Layers**	**Hidden size**	**Heads**	**Batch size**	**Epochs**
uni-LSTM	1	64	-	64	10
bi-LSTM	2	64	-	64	10
LSTM seq2seq	3	64	-	64	10
Transformer seq2seq	6	64	4	64	10

Table 4.7: Summary of the all the model parameters used for producing the results when training models with HDFS datasets.

4.3.1 Uni-LSTM model

As mentioned before, the results are from the models' corresponding best thresholds that yield the best F1 score.

Mixed training dataset

The best thresholds for training data proportion 1%, 5%, 10% and 100% are 4, 4, 4, and 5, respectively. Figure 4.4 shows the results of training different ratio of the mixed training data on the Uni-LSTM model, and we observed that the model trained with 1% of the mixed training data performs best.

In addition, when we increase the proportion of the training data from 10 % to 100%, the precision increased and the recall decreased, which means when we use more training data to train the model, the network will predict less anomalous predictions.

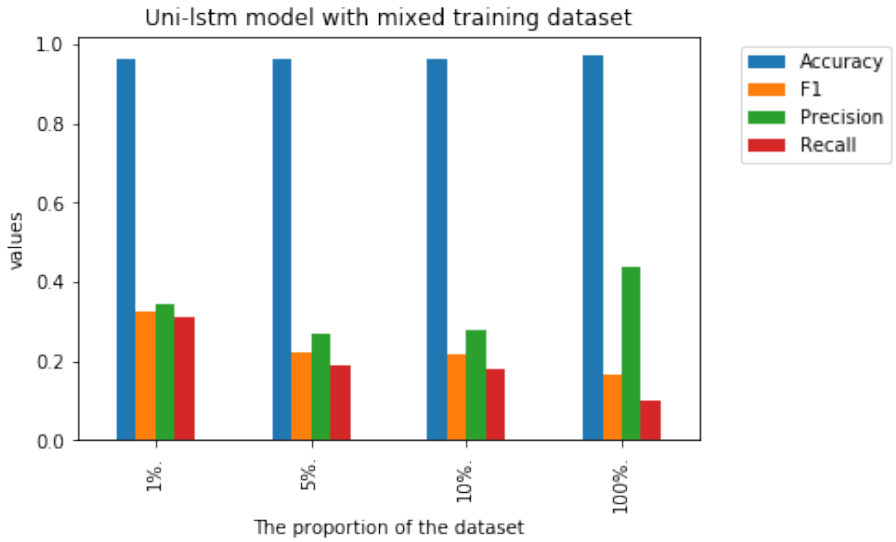

Figure 4.4: Results of the uni-LSTM based anomaly detectors for various proportions of the mixed training data on the HDFS data.

Normal training dataset

The best threshold for training data proportion 1%, 5%, 10% and 100% are 7, 6, 7, and 6, respectively. Figure 4.5 shows the results of different ratio of the normal training data on the Uni-LSTM model, and we observed that the model trained with 100% of the normal training data performs best. Furthermore, all proportions of the training dataset performed well.

Figure 4.5: Results of the uni-LSTM based anomaly detectors for various proportions of the normal training data on the HDFS data.

4.3.2 Bi-LSTM model

As mentioned before, the results are from the models' corresponding best thresholds that yield the best F1 score.

Mixed training dataset

The best threshold for the training dataset proportion 1%, 5%, 10% and 100% are 3, 3, 3, and 5, respectively. Figure 4.6 shows the results of different size of the normal training data on the Bi-LSTM model, and we observed that the model trained with 1% of the mixed training data performs best.

Similar observation as the Uni-LSTM with mixed training dataset, when we increase the proportion of the training data from 10 % to 100%, the precision increased and the recall decreased, which means the network will predict less anomalous predictions when we use more training data to train.

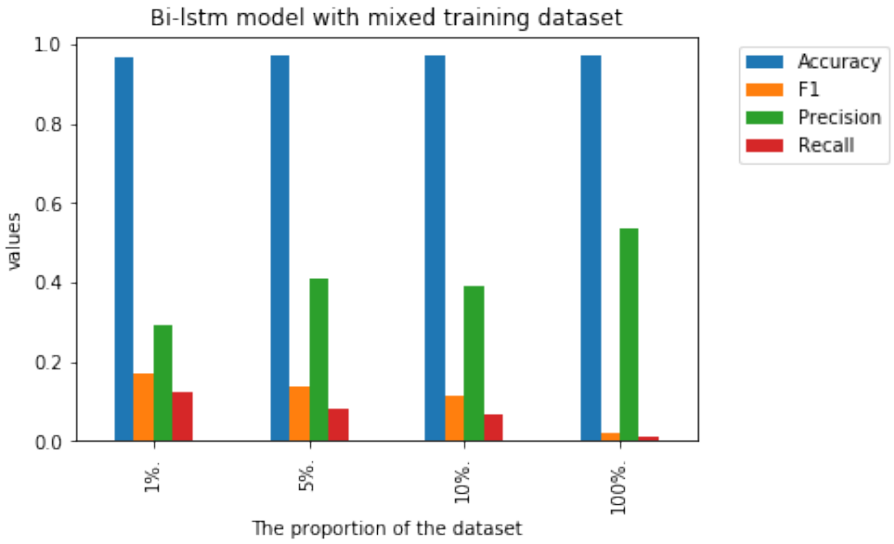

Figure 4.6: Results of the bi-LSTM based anomaly detectors for various proportions of the mixed training data on the HDFS data.

Normal training dataset

The best threshold for training dataset proportion 1%, 5%, 10%, and 100% are 5, 4, 4, and 4, respectively. Figure 4.7 shows the results of different sizes of the normal training data on the Bi-LSTM model, and we observed that the model trained with 5% of the normal training data performs best.

In addition, the results of 1%, 5%, and 10% proportions of the training data are quite similar. However, when we increase the proportion of the training dataset from 10% to 100%, almost similar precision but recall decreased, this means more false negatives were detected.ed.

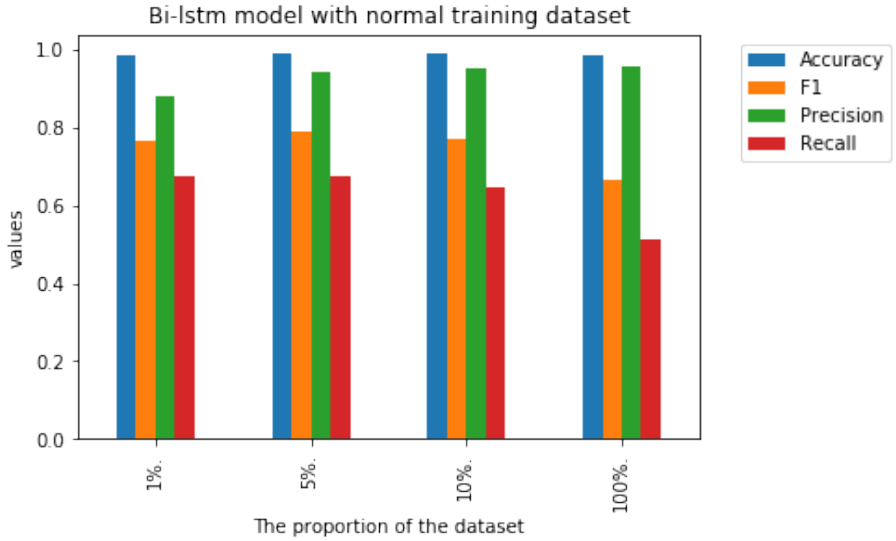

Figure 4.7: Results of the bi-LSTM based anomaly detectors for various proportions of the normal training data on the HDFS data.

4.3.3 LSTM sequence-to-sequence model

When evaluating the anomaly detection performance of the LSTM sequence-to-sequence model the result was that only very few log files were ever predicted correctly. The minimum accuracy for one log file was 0, the maximum accuracy for one log file was 1 (very few). The anomaly detection average accuracy of the mixed dataset is 0.2876, and of the normal dataset is 0.5780. In this case, we can not do any effective anomaly detection by using this model on both mixed training dataset and normal dataset.

4.3.4 Transformer sequence-to-sequence model

Mixed training dataset

The 100% dataset here contains 3817328 windows. Figure 4.8 shows the results of the transformer model with training on the mixed dataset with different proportions, and we observed that the model trained with 1% of the mixed training data performs best.

Moreover, we observed that when we increased training data from 1% to 100%, the F1 score will decrease to almost zero. This observation presents that the model can find very few anomalies if the model trained with 100% training data.

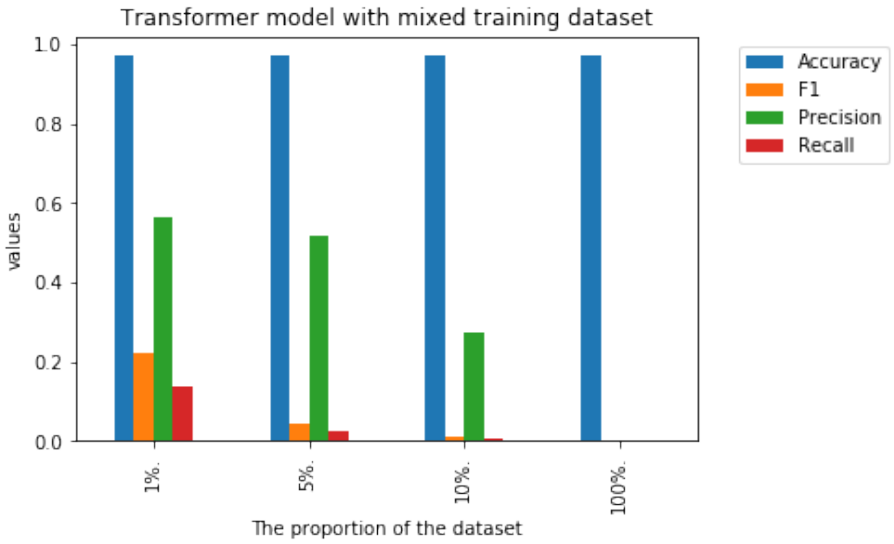

Figure 4.8: Results of the Transformer based anomaly detectors for various proportions of the mixed training data on the HDFS data.

Normal training dataset

The 100% dataset here contains 3750414 windows. Figure 4.9 shows the results of the transformer model with training on the normal dataset with different proportions, and we observed that the model trained with 1% of the normal training data performs best.

Significantly, the recall is 1 when we trained with 1% of the normal training data which means the model caught all the true positives. And the precision is 1 when we trained with 100% of the normal training data which means no false positive in this case.

Figure 4.9: Results of the Transformer based anomaly detectors for various proportions of the normal training data on the HDFS data..

4.3.5 Comparing models

The previous subsections show the results of different proportions of the training data in each model. To find the best model for the mixed/normal dataset, we choose the best ratio of the training data for each model according to the highest F1 score then compare them. Figure 4.10 and Figure 4.11 present the best proportion of the training data for each model for both the mixed dataset and the normal dataset, and provide some observations.

For the mixed training data: smaller proportions of training data always get better results, and 1% is the best proportion of the training data for all three models.

For the normal training data: the best proportion of the data for the uni-LSTM model is 100%, for the uni-LSTM model is 5%, and for the transformer model is 1%. Moreover, the transformer model with 1% proportion of the training data has recall 1 but lower precision causes a lower F1 score.

Overall, the results of training on the normal training data are always better than the results of training on the mixed training data, and the uni-LSTM model performs best for both the mixed training dataset and the normal dataset. Besides, the bi-LSTM model seems no advantage compare to the other two models neither in the mixed training data nor the normal training data.

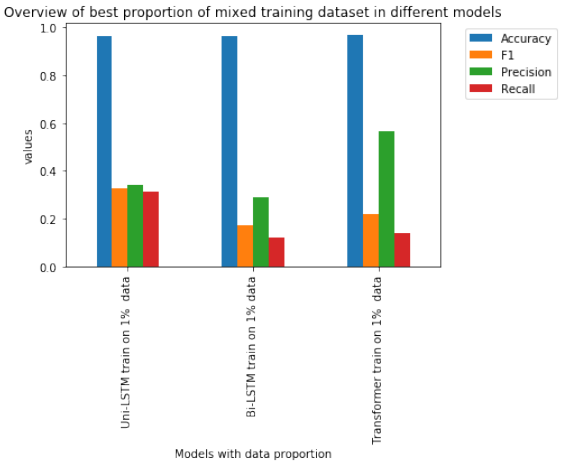

Figure 4.10: Best result for each model with mixed data on the HDFS data.

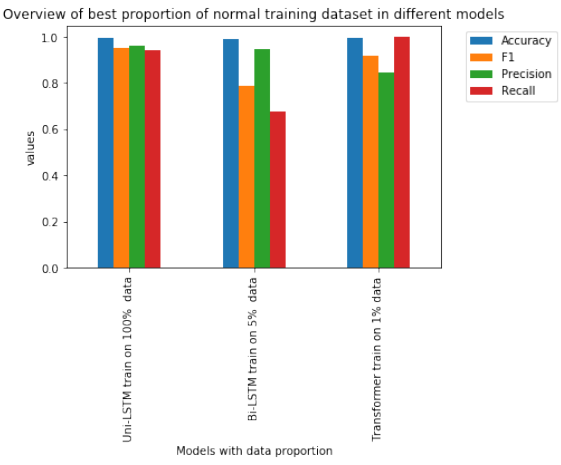

Figure 4.11: Best result for each model with normal data on the HDFS data.

5
Discussion and Conclusion

5.1 Discussion

Out of the anomaly detection approaches investigated in this project, the uni-LSTM model with a top-n threshold and the Transformer model trained on 1% of the training data perform the best on the Volvo GTT data and the uni-LSTM model with a threshold trained on 100% of the normal training data perform the best on the HDFS data. The disadvantage of the uni-LSTM method is the use of the threshold, which needs to be found with trial-and-error. Further, for the Volvo GTT data it is not certain how this method will perform in practice due to the small imbalanced data set the anomaly detection methods have been tested on. However, as the uni-LSTM method successfully pointed out the anomalous logs for all proportions of data the results indicate that this method might be used for filtering out logs which may automate the anomaly detection process to some extent.

Moving on to the HDFS data, the best results are based on training with only normal data while the mixed data yield poor results. The uni-LSTM method also gets the best result for the training on mixed data compares to other methods, but the F1-score is low indicating poor performance. The results from using the HDFS data show that the results of training on the dataset containing no anomalies are consistently better than the results of training on the mixed dataset which includes anomalies. To reiterate, training on a data set containing no anomalies is how most other approaches implemented their anomaly detection models.

Another observation is the bi-LSTM model performs worse than the uni-LSTM model. The issue may when we created the inputs for the bi-LSTM model the bottom information is missing, if the anomalies exist at the bottom of the log file then the bi-LSTM model has no chance to indicate them. The issue could also be an effective way to read the information is to read from one direction. But in the bi-LSTM model, we read the information from both directions.

We found no significant advantage to adding more recurrent layers in the uni- and bi-LSTM networks. Neither did the methods improve when we used a smaller or higher dimension for the hidden states. Lastly, in DeepLog [7] two layers were utilized in the LSTM neural network. This method did not perform better than a one-layer LSTM network when using the Volvo GTT data. Thus, as we noticed no improvement in the anomaly detection we chose to only use one LSTM-layer in the

architectures to decrease the time it takes to train the network. It might be that replicating the DeepLog method exactly would produce better results for the HDFS data. However, the aim of this project is to find a method that works well on the Volvo GTT data and therefore we used the same methods for the Volvo GTT data and the HDFS data.

Moving on to the LSTM sequence-to-sequence model, the issue might be that we are not using a top-n threshold here to compute the n most likely log keys in the output. The reason why we wanted to try this approach was to see how methods without a threshold perform. Comparing the results from the sequence-to-sequence LSTM model to the sequence-to-sequence Transformer model, we can see that the Transformer is better at learning the log sequences. This result is supported by the fact that Transformers perform better in neural machine translation [31] tasks.

Moreover, our results point to the fact that changing the size of the dataset might influence the performance of the anomaly detector. A more thorough analysis of this would be interesting to implement. For the Volvo GTT data all the models performed better with a smaller amount of data. Meanwhile for the mixed HDFS dataset training on a smaller proportion of data was also better but the F1-score was small for all models. When removing the anomalies from the HDFS data (the normal dataset) the uni-LSTM model performed better with a larger amount of data while the bi-LSTM model and Transformer performed better with a smaller amount of data.

5.2 Future work

As mentioned in section 1.4, there are other ways than regex to pre-process the data. Pre-defined log parsers exist such as Spell [8] and Drain [12] and it would be interesting to try a different data preprocessing approach with a pre-defined log parser. Aside from this, it is difficult to identify the areas in which the anomaly detection methods could be improved when using an unsupervised approach. However, it seems likely that training the models using only confirmed non-anomalous data for the Volvo GTT dataset will increase the performance. This was unfortunately not possible during this project, but it would be interesting to further evaluate how the methods would perform in this scenario.

In its current states, the sequence-to-sequence model (section 3.3.3) and the Transformer (section 3.3.4) trained on the entire training data set is not usable. The sequence-to-sequence model does not reconstruct the log line windows well, while the Transformer is too good at reconstructing the window of log lines. Adding an attention mechanism to the LSTM sequence-to-sequence model would be interesting to see if this is a middle-ground to this issue. Also, using strict teacher forcing during training might be misguiding for the model.

Furthermore, other problem definition approaches to the sequence-to-sequence models could be considered, such as predicting more than one log line based on some context of past log lines which might introduce some more complexity to the Transformer model. Additionally, it might be interesting to get rid of the sliding windows approaches and try to reconstruct an entire log sequence.

Also, in the research where the uni-LSTM method was inspired by [7] the neural networks are trained on a set of only normal (non-anomalous) data. It might be that when including abnormal log sequences during the training, the networks learn these too well and the assumption that they are too few to be learned do not hold. This is an issue when the data set is so big that something that is uncommon in a couple of weeks might not be so uncommon when accumulated over a few months. The network then receives these inputs and no consideration of the occurrences with respect to time is done. On the flip side, one month might be too little to capture the behaviour of the data as one month might be some exception to the normal behaviour.

5.3 Conclusion

The results from the experiments run on the Volvo GTT data set do not provide a definite conclusion to be made regarding the anomaly detection performance due to the anomaly detection data set being small and imbalanced. The best models yielded 8 true positives and 0 false negatives with 7 true positives. We do not know how this will scale with a larger data set. However, in the results run on the HDFS data the performance quite vastly differs between using mixed data and data sets using only normal data. The models using only 'normal' data with the HDFS data set provide quite good results. However, it is not clear how this is relatable to a real-life application with large data sets and no labels in the data set.

Assuming that we have some system producing a lot of log files which may change slightly over time and require retraining within intervals, manually labelling all these log files will too result in time-consuming work. It is also not completely clear why one would choose an unsupervised approach in this case, as this might as well be used as a supervised classification task if the labels already exist because the method require only normal data.

Furthermore, the labelling of the files in order for it to be used as a semi-supervised approach is quite unclear. A log file might contain an anomaly that the user *wants* to output (for example an error message) but this might not be abnormal behaviour in the large data set. On the opposite hand, the user might not consider a log file as anomalous but in the context of the data set it is uncommon and an anomaly. Therefore, using the approach presented in this thesis, the labelling of the data would also require an analysis of the data set. The anomaly detection method might output some anomalies, but it also outputs log files with a process breaking down that the user does not consider an anomaly.

In conclusion, the results from the anomaly detection performed on the HDFS dataset seem to indicate that using a semi-supervised approach using only data that is confirmed non-anomalous seem to be critical when creating an anomaly detector for this dataset. Compared to the dataset used for anomaly detection on the Volvo GTT dataset these results could be viewed as more reliable as there are more labeled log sequence to use for anomaly detection. Therefore, a definite conclusion cannot be made of how the anomaly detectors work in practice on the Volvo GTT data. However, the Volvo GTT dataset is larger and has a bigger dictionary size than the HDFS dataset which might influence the suitability of an unsupervised approach. As a final remark, the uni-LSTM model does successfully point out all anomalies in the Volvo GTT data for all proportions of data, which indicate that this method might be used for filtering out logs that are not of interest without missing the anomalies.

Bibliography

[1] Andrej Karpathy blog. (2015, May 21) *The Unreasonable Effectiveness of Recurrent Neural Networks [Blog post].* Retrieved from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[2] Bahdanau, D., Cho, K., Bengio, Y. (2016) *Neural Machine Translation by Jointly Learning to Align and Translate.* ICLR 2015. arXiv:1409.0473v7.

[3] Bertero, C., Roy, M., Sauvanaud, C., Tredan, G. (2017). *Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection.* 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). https://doi.org/10.1109/issre.2017.43

[4] Chollet, F. et al. (2015) *Keras* https://github.com/fchollet/keras

[5] Colah's blog. (2015, August 27) *Understanding LSTM Networks [Blog post].* Retrieved from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[6] Dai, H., Li, H., Chen, C. S., Shang, W., Chen, T.-H. (2020). *Logram: Efficient Log Parsing Using n-Gram Dictionaries.* IEEE Transactions on Software Engineering. https://doi.org/10.1109/tse.2020.3007554

[7] Du, M., Li, F., Zheng, G., Srikumar, V. (2017). *Deeplog: Anomaly Detection and Diagnosis from System Logs through Deep Learning.* Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. https://doi.org/10.1145/3133956.3134015

[8] Du, M., Li, F. (2016). *Spell: Streaming Parsing of System Event Logs.* 2016 IEEE 16th International Conference on Data Mining (ICDM). https://doi.org/10.1109/icdm.2016.0103

[9] Fu, Q., Lou, J.-G., Wang, Y., Li, J. (2009). *Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis.* 2009 Ninth IEEE International Conference on Data Mining. https://doi.org/10.1109/icdm.2009.60

[10] Goldberg,Y., Hirst, G. (2017). *Neural Network Methods in Natural Language Processing (Synthesis Lectures on Human Language Technologies).* Morgan

Claypool Publishers.

[11] Hastie, T. et al., *The Elements of Statistical Learning,* Second Edition, 485,Springer Science+Business Media, LLC 2009. DOI: 10.1007/b94608_14.

[12] He, J. Zhu, Z. Zheng and M. R. Lyu.(2017) *Drain: An Online Log Parsing Approach with Fixed Depth Tree.* 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 2017, pp. 33-40, doi: 10.1109/ICWS.2017.13.

[13] He, S. et al. (2020) *Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics.* arXiv:2008.06448v1.

[14] He, S., Zhu, J., He, P., Lyu, M. R. (2016) *Experience Report: System Log Analysis for Anomaly Detection*, IEEE International Symposium on Software Reliability Engineering (ISSRE), 2016. [Bibtex][] (ISSRE Most Influential Paper). DOI: 10.1109/ISSRE.2016.21.

[15] Huang, S. et al (2020) *HitAnomaly: Hierarchical Transformers for Anomaly Detection in System Log.* IEEE Transactions on Network and Service Management PP(99):1-1. DOI: 10.1109/TNSM.2020.3034647

[16] Kingma, D. P., Ba, J. L. (2017) *Adam: A Method for Stochastic Optimization.* Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. arXiv:1412.6980v9.

[17] Lyu, R. M., Zhu, J., He, P., He, S., Liu, J. (2019) *LogPAI.* Retrieved from http://www.logpai.com/. Github: https://github.com/logpai

[18] Liu, Z., Qin, T., Guan, X., Jiang, H., Wang, C. (2018). *An Integrated Method for Anomaly Detection From Massive System Logs.* IEEE Access, 6, 30602–30611. https://doi.org/10.1109/access.2018.2843336

[19] Lipton, Z. C. (2018). *The Mythos of Model Interpretability.* Queue, 16(3), 31–57. https://doi.org/10.1145/3236386.3241340

[20] Luong, M-T, Pham, H., Manning, C. D. (2015) *Effective Approaches to Attention-based Neural Machine Translation.* arXiv:1508.04025v5

[21] Makanju, A., Zincir-Heywood, A. N., Milios, E. E. (2012). *A Lightweight Algorithm for Message Type Extraction in System Application Logs.* IEEE Transactions on Knowledge and Data Engineering, 24(11), 1921–1936. https://doi.org/10.1109/tkde.2011.138

[22] Malhotra, P. et al (2016) *LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection.* Accepted at ICML 2016 Anomaly Detection Workshop,

New York, NY, USA, 2016. arXiv:1607.00148v2

[23] Mehlig, B. (2021) *Machine learning with neural networks.* arXiv:1901.05639v3 [cs.LG] 10 Feb 2021

[24] Meng, W., Liu, Y., Zhu, Y., Zhang, S., Pei, D., Liu, Y., Chen, Y., Zhang, R., Tao, S., Sun, P., Zhou, R. (2019). *LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs.* Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. https://doi.org/10.24963/ijcai.2019/658

[25] Morphy, E. (2018, May 31). *What Is a Neural Network and How Are Businesses Using Them?* CMSWire.Com. https://www.cmswire.com/digital-experience/what-is-a-neural-network-and-how-are-businesses-using-it/

[26] Nedelkoski, S. et al (2020) *Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs* 20th IEEE International Conference on Data Mining. arXiv:2008.09340v1

[27] Paszke, Adam et al. (2017). *Automatic differentiation in PyTorch*

[28] Pande, A., Ahuja, V. (2017). WEAC: Word embeddings for anomaly classification from event logs. 2017 IEEE International Conference on Big Data (Big Data). https://doi.org/10.1109/bigdata.2017.8258034

[29] Reddi, S. J. , Kale, S., Kumar, S. (2019) *On the Convergence of Adam and Beyond*. Appeared in ICLR 2018. arXiv:1904.09237v1

[30] Sutskever, I., Vinyals, O., Le, V. Q. (2014) *Sequence to Sequence Learning with Neural Networks*. NIPS'14: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2. arXiv:1409.3215v3

[31] Vaswani, A. et al (2017) *Attention is All You Need*. arXiv:1706.03762v5.

[32] Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, Zhen Ming Jiang. (2014) *Understanding Log Lines Using Development Knowledge.* 2014 IEEE International Conference on Software Maintenance and Evolution.

[33] Xu, W. et al (2010) *Detecting Large-Scale System Problems by Mining Console Logs*. Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel. DOI:10.1145/1629575.1629587

[34] S. Zhang, S. M. H. Bamakan, Q. Qu and S. Li, *Learning for Personalized Medicine: A Comprehensive Review From a Deep Learning Perspective* IEEE Reviews in Biomedical Engineering, vol. 12, pp. 194-208, 2019, doi:

10.1109/RBME.2018.2864254.

[35] Zhang, X. et al (2019) *Robust Log-Based Anomaly Detection on Unstable Log Data*. ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. DOI: https://doi.org/10.1145/3338906.3338931

[36] Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., Lyu, M. R. (2019). *Tools and Benchmarks for Automated Log Parsing.* 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 121–130. https://doi.org/10.1109/icse-seip.2019.00021

[37] Edupristine. 2018. Steps to build a Predictive Modeling. 9th June. Steps to build a Predictive Modeling. [Online]. [26 April 2021]. Available from: https://www.edupristine.com/blog/predictive-modeling-steps

[38] Hosted by LogPAI Team. (2018). Loghub [Data set]. Zenodo. The raw logs can be requested from Zenodo: https://doi.org/10.5281/zenodo.1144100.