CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER'S THESIS

Anomaly Detection Methods for Log Files

Author:
Bc. Martin Koryťák

Supervisor:
Ing. Jan Drchal, Ph.D.

Open Informatics
Department of Computer Science
Faculty of Electrical Engineering

May, 2021

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Koryťák Martin** Personal ID number: **457010**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Data Science**

II. Master's thesis details

Master's thesis title in English:

Anomaly Detection Methods for Log Files

Master's thesis title in Czech:

Metody detekce anomálií pro soubory logů

Guidelines:

The task is to experiment with methods of anomaly detection applied to log files.
While most current approaches deal with parsed log templates and their categorical distributions, this work will focus on continuous vector representations (real embedding vectors).
1) Familiarize yourself with state-of-the-art anomaly detection methods focusing on processing log files as well as on continuous vector representations.
2) Select a set of existing methods and compare their performance on log anomaly detection. The supervisor will supply log embedding methods such as [5].
3) Benchmark datasets will be supplied by the supervisor: HDFS, BGL, OpenStack (see https://github.com/logpai/loghub) are possible choices.

Bibliography / sources:

[1] He, Shilin, et al. "Experience report: System log analysis for anomaly detection". 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2016.
[2] Chalapathy, Raghavendra, and Sanjay Chawla. "Deep learning for anomaly detection: A survey."; arXiv preprint arXiv:1901.03407 (2019).
[3] Chandola, Varun, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey"; ACM computing surveys (CSUR) 41.3 (2009): 1-58.
[4] Du, Min, et al. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning." Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017.
[5] Marek Souček, "Log Anomaly Detection", master thesis, supervisor Jan Drchal, FEE CTU, 2020.

Name and workplace of master's thesis supervisor:

Ing. Jan Drchal, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **12.02.2021** Deadline for master's thesis submission: _____

Assignment valid until: **30.09.2022**

Ing. Jan Drchal, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to express my sincerest gratitude and acknowledgment to my supervisor Ing. Jan Drchal, Ph.D., for the guidance and helpful advice. Furthermore, I would like to thank my family, who have been fully supportive throughout my studies and everything I ever wanted to do. Last but not least, huge heartfelt thanks go to my alma mater, Czech Technical University, for providing me with all necessary education and being instrumental in gaining valuable experience abroad.

Author Statement for Graduate Thesis

I, Martin Koryťák, declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Signed: _____

Date: _____

Abstract

This thesis is dedicated to methods of anomaly detection applied to log files. The current state-of-the-art anomaly detection methods usually follow the traditional approach for log processing. Firstly, log files are processed by a log parsing technique which transforms text information into non-specific structured data. Next, the data is converted into a numerical representation. The feature extraction is often related to natural language processing techniques. However, the traditional approach requires extensive domain knowledge and retraining a particular model when new log messages become available. Thanks to the recent advancements in the natural language processing domain, we can directly learn embedding vectors instead of the feature extraction based on log parsing. We propose novel autoencoder-based models leveraging the embedding vectors since autoencoders are a recommended choice in the field of anomaly detection. Moreover, we experiment with various techniques which are incorporated into autoencoders, such as convolutional layers and the self-attention mechanism. We verify that the autoencoders utilizing convolutional layers are effective for anomaly detection in log files. Furthermore, we demonstrate that boosting the models with the self-attention mechanism might be advantageous and open room for future work and further research. Finally, we can conclude that the traditional approach combined with an autoencoder may achieve impressive results on the provided testing data set. Nonetheless, the AECNN1D model achieves the most promising results among models leveraging the embedding representation of logs — the F1-score is 0.8597 on the testing data set. The AECNN1D model is generally applicable to deploying into the production since no additional requirements or periodic retraining is necessary.

Keywords: anomaly detection, natural language processing, log files, autoencoder, convolutional neural network, machine learning

Abstrakt

Tato práce se věnuje metodám detekce anomálií aplikovaným na soubory logů. Současné metody detekce anomálií obvykle používají tradiční přístup ke zpracování logů. Nejprve se soubory logů zpracují jejich parsováním, které transformuje textové informace na nespecifická strukturovaná data. Poté jsou data převedena na číselnou reprezentaci. Extrakce příznaků často souvisí s technikami používanými pro zpracování přirozeného jazyka. Tradiční přístup však vyžaduje rozsáhlé oborové znalosti a přeučení modelu, když se objeví nové typy logů. Díky nedávným pokrokům v oblasti zpracování přirozeného jazyka můžeme přímo naučit vnoření slov namísto extrakce příznaků založené na parsování logů. Navrhujeme nové modely založené na autoenkodérech využívajících vnoření slov, protože jsou doporučovanou volbou v oblasti detekce anomálií. Kromě toho experimentujeme s různými technikami, které jsme začlenili do autoenkodérů, jako jsou konvoluční vrstvy a mechanismus self-attention. Ověřujeme, že autoenkodéry využívající konvoluční vrstvy jsou vhodné pro detekci anomálií v souborech logů. Dále ukazujeme, že přidání mechanismu self-attention do modelů může být výhodné a otevírá prostor pro budoucí práci a další výzkum. Závěrem můžeme konstatovat, že tradiční přístup v kombinaci s autoenkodérem může na poskytnuté testovací datové sadě dosáhnout působivých výsledků. Nicméně model AECNN1D dosahuje nejslibnějších výsledků mezi všemi modely, které využívají vnoření slov logů — metrika F1-score je 0,8597 na testovací datové sadě. Model AECNN1D je obecně použitelný pro nasazení do produkce, protože nemá žádné další požadavky ani nevyžaduje občasné přeučování.

Klíčová slova: detekce anomálií, zpracování přirozeného jazyka, soubory logů, autoenkodér, konvoluční neuronová síť, strojové učení

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is dedicated to methods of anomaly detection applied to log files. The current state-of-the-art anomaly detection methods usually follow the traditional approach for log processing. Firstly, log files are processed by a log parsing technique which transforms text information into non-specific structured data. Next, the data is converted into a numerical representation. The feature extraction is often related to natural language processing (NLP) techniques. However, the traditional approach requires extensive domain knowledge and retraining a particular model when new log messages become available. Thanks to the recent advancements in the NLP domain, we can directly learn embedding vectors instead of the feature extraction based on log parsing. We researched semi-supervised and unsupervised machine learning methods applicable for anomaly detection of semi-structured data, such as log files. We decided to leverage autoencoder-based models as autoencoders are a recommended choice in the field of anomaly detection.

1.1 Motivation

Anomaly detection methods are daily used in a variety of domains across industries. Anomaly detection methods are instrumental in the banking sector, protecting us against fraudulent activity and transactions. In the cybersecurity domain, these methods help researchers to detect attackers' sophisticated malicious software. The methods also help scientists and doctors to detect cancer or chronic illness in the healthcare industry. Last but not least, anomaly detection methods monitor the behavior of a distributed system in a data center and detect an anomalous state of a particular machine, which is the scope of this thesis.

With the current demand for cloud services, it has become increasingly important to provide highly available and error-free services. High availability of services can be achieved by quickly identifying the root cause. Furthermore, log files are one of the possible solutions. Nowadays, no company can afford to inspect the log files manually. Therefore, there is a surge of interest in developing methods which can automatically detect anomalous behavior with high accuracy.

1.2 Thesis structure and aims

The fundamental aim of this thesis is to experiment with methods of anomaly detection applied to log files. Specifically, Chapter 3 delineates state-of-the-art anomaly detection methods. Moreover, this chapter further contains a detailed overview of the methods focusing on processing log files and extracting continuous vector representation of features. Next, Chapter 4 is dedicated to selecting various existing

methods and comparing their performance on a provided data set with our proposed models. The comprehensive evaluation of all models on the data set can be found in Chapter 5.

This thesis is divided into individual chapters. Chapter 2 introduces and defines the terms and theoretical concepts used in the following chapters. Chapter 3 is devoted to the related work and the classification of anomalies and approaches for log processing. The detailed solution approach can be found in Chapter 4. Furthermore, this chapter also delineates the implementation of proposed models and algorithms. Chapter 5 contains a comprehensive evaluation of both baseline and proposed models on the data set. This chapter is concluded with an exhaustive discussion of achieved results. Chapter 6 closes the whole thesis with final remarks and further directions for future work.

Chapter 2

Fundamentals

This chapter explains and defines the terms and theoretical concepts used in the following chapters. We start with a feedforward neural network. Next, we introduce the concept of a convolutional neural network and the temporal convolutional network. We also mention the self-attention mechanism, which is a crucial part of the state-of-the-art architectures in the NLP domain. Finally, we present how we can leverage neural networks in semi-supervised machine learning.

2.1 Neural network

The biological analogy of animal brains inspires neural networks. Akin to a brain, the initial building block of all neural networks is a neuron; see Figure 2.1. Each neuron accepts several input values and produces one output value. The core idea behind the neuron is that it is possible to learn a particular weight w_i given by an individual input x_i. The simplest form of a neural network is called the perceptron [1].

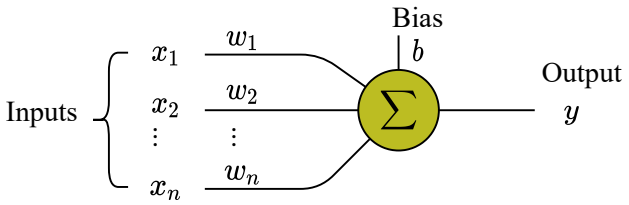


FIGURE 2.1: An example of a neuron, a building block of a neural network. Concatenation of many neurons and organizing them into layers can produce a very complex neural network.

A feedforward neural network contains multiple neurons which comprise a hidden layer. This hidden layer may also be called a fully connected layer if each neuron from the previous layer l_i is connected to each neuron from the succeeding layer l_{i+1}. The input layer receives an input, and the output layer makes a prediction. An example of a feedforward neural network is depicted in Figure 2.2.

Neural networks use learning techniques — a well-known technique is backpropagation. A loss function computes the loss using the output values and the correct answers. Next, the learning technique adjusts the weights of individual neurons. This process decreases the loss until it plateaus.

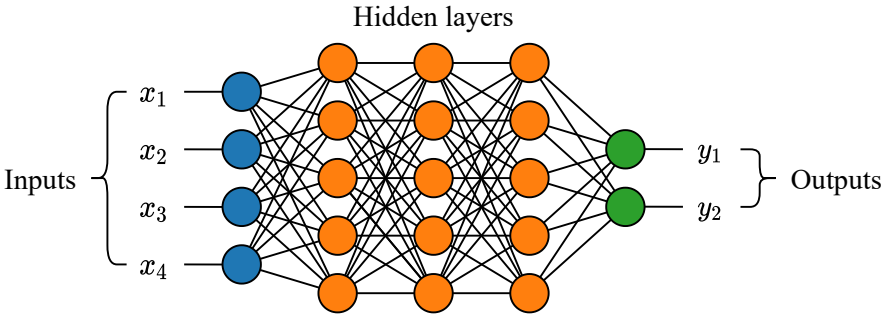


FIGURE 2.2: A classic example of a feedforward neural network. This neural network comprises only fully connected layers, which contain trainable parameters.

2.2 Convolutional neural network

Neural networks consist of individual neurons, which may include various connections and stacking. Convolutional neural networks have a known grid-like topological structure [2]. Traditionally, recurrent neural networks process time-series data, but they are difficult to train and parallelize [3], and thus they are often superseded by other approaches. Time-series data may also be processed using a 1D convolutional neural network, whereas a 2D convolutional neural network may be applicable for imagery data.

2.2.1 Building blocks

Each convolutional neural network leverages convolutional layers, pooling layers, and fully connected layers. We delineate all mentioned building blocks below.

Convolutional layer

A convolutional layer is the main building block, and the whole architecture is named after this layer. Each neuron is connected to particular neurons from the preceding layer (a receptive field), which significantly reduces the number of trainable parameters. The reduction of trainable parameters usually allows the architecture of a convolutional neural network to be deeper. This layer is used chiefly in the first part of a convolutional neural network, and it is responsible for feature extraction. The essential hyperparameters are the kernel size and the number of feature maps. Each feature map represents one output matrix after applying a convolutional kernel on a particular input. Padding, stride, and dilation are additional hyperparameters of a convolutional layer. The difference between the 1D convolutional layer and the 2D convolutional layer is depicted in Figure 2.3.

Precisely, an arbitrary convolutional layer is defined in Equation 2.1, where n denotes the number of input channels, and i denotes the index of a particular output channel. The symbol $*$ is the discrete convolution operator.

$$outputs_i = biases_i + \sum_{j=1}^{n} weights_{i,j} * inputs_j \tag{2.1}$$

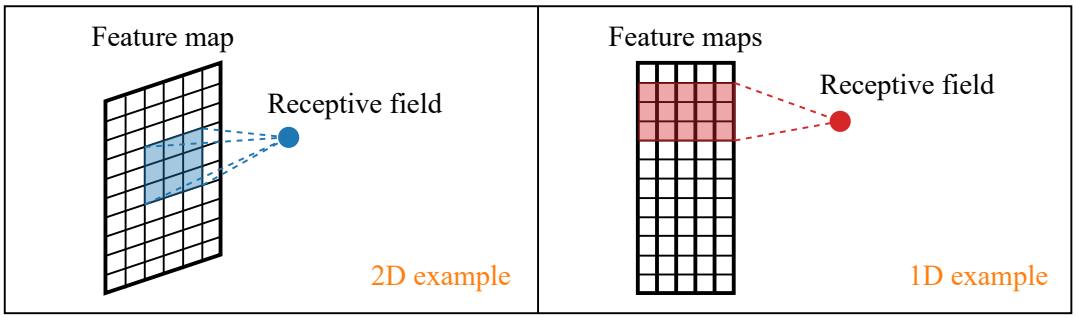


FIGURE 2.3: An example of convolutional layers with different input shapes. The kernel convolves across the input and produces a new feature map.

Transposed convolutional layer

A transposed convolutional layer has been developed to upsample an input, i.e., an output feature map has greater spatial dimensions than the input. The transposed convolutional layer is defined as the inverse operation of the convolutional layer. However, applying $tconv(conv(X))$ might not produce a new matrix with the same numbers as the original input matrix X. The operations $conv$ and $tconv$ denote convolutional and transposed convolutional layers, respectively. Specifically, the operation of the 2D transposed convolution is depicted in Figure 2.4.

The transposed convolutional layer can learn kernel weights. Next, the trained kernel upsamples the input feature maps. On the contrary, there exists an upsampling technique which only copies the surrounding numerical values, i.e., it does not provide any trainable parameters. Akin to the convolutional layer, there exist various transposed convolutional layers which operate with different input shapes. The crucial hyperparameters are the number of feature maps, the kernel size, padding, stride, and dilation.

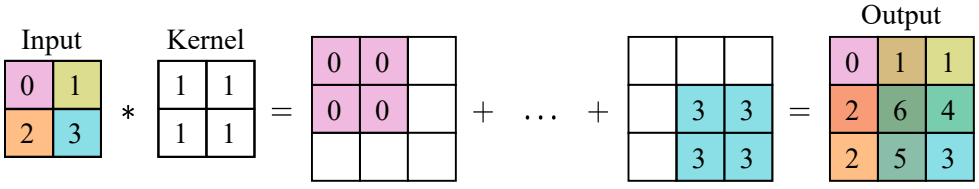


FIGURE 2.4: An example of the 2D transposed convolutional layer. The kernel convolves across the input matrix, and it produces four intermediate results, which are finally added together. The opacity of individual colors was set to 50% in order to capture the impact of each intermediate matrix on the output matrix. The symbol $*$ is the discrete convolution operator.

Pooling layer

A pooling layer is usually inserted between two convolutional layers, which leads to the reduction of trainable parameters. Models with less trainable parameters better generalize on unseen data examples. There are various types of pooling layers, such as max-pooling and average pooling layers. An example of these layers is shown in Figure 2.5. Since a particular pooling layer applies an aggregation function to the input, the pooling layer does not have any trainable parameters.

A global pooling layer can be leveraged as the output layer instead of a fully connected layer before applying the softmax activation function [4, 5].

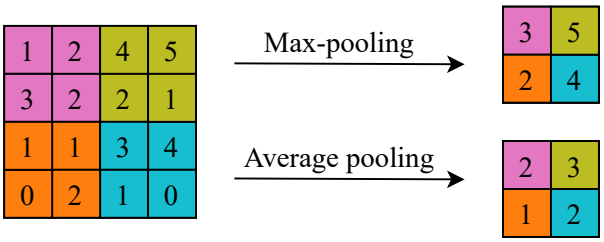


FIGURE 2.5: An example of various pooling layers with the stride equals two. A pooling layer downsamples an input feature map. The usual choice of pooling layers is the max-pooling layer which reduces the number of trainable parameters and decreases the risk of overfitting.

Fully connected layer

A fully connected layer comprises individual neurons which are connected to each output of the preceding layer. A convolutional neural network leverages fully connected layers at the end of the network since they exponentially increase the complexity of a particular architecture. Training a complex network containing only fully connected layers might be slow and prone to overfitting.

Regularization

The authors [6] proposed a regularization technique called dropout. Dropout is recommended as a technique tackling with overfitting of a model. The main idea is to randomly deactivate neurons in a particular neural network during the training phase. This prevents neurons from precisely adapting to the input. On the contrary, all neurons are active during the testing phase. However, their weights are multiplied by the probability p of deactivation of the neurons. Researchers usually insert dropout in-between two fully connected layers. It leads to the depletion of a particular fully connected layer after applying the dropout technique; see Figure 2.6.

Besides dropout, there exist other regularization techniques such as L1 and L2 regularizations. L1 regularization adds the absolute value of a weight w multiplied by the regularization strength λ to the computed loss. L1 regularization has its property which leads to sparse weight vectors. On the other hand, L2 regularization adds a weight squared w^2 multiplied by the regularization strength $\frac{1}{2}\lambda$ to the computed loss. L2 regularization penalizes jagged weight vectors [7]. The authors [8] combined both L1 regularization and L2 regularization, i.e., the proposed regularization is $\lambda_1|w| + \lambda_2 w^2$.

Activation function

An activation function applies a mathematical function on a particular output of a neuron — an example of the activation function ReLU is depicted in Equation 2.2. A traditional choice of activation functions used to be the sigmoid function. The sigmoid function suffers from the saturating gradient problem, i.e., in both tails of the curve, the gradient is close to zero. Therefore, the careful initialization of weights is required in order to prevent saturation [9]. Moreover, the sigmoid function is

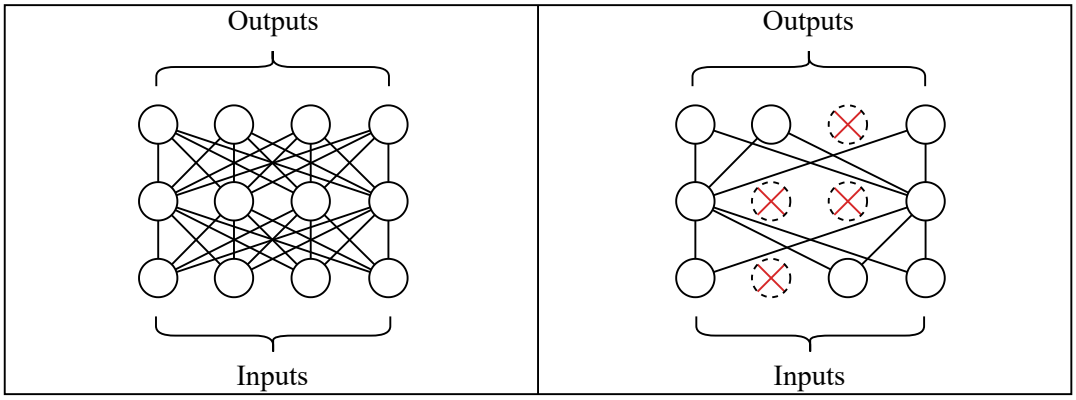


FIGURE 2.6: An example of a network with fully connected layers (left) and after applying dropout (right).

not zero-centered, which might introduce undesirable dynamics in the gradient updates [7]. The hyperbolic tangent is akin to the sigmoid function, but it is a zero-centered function. The activation function ReLU is used nowadays, and its new modifications have been recently proposed, such as Leaky ReLU [10], ELU [11], and SELU [12].

$$output = \texttt{max}(0, bias + \sum_{i=1}^{n} weights_i \cdot inputs_i) \tag{2.2}$$

2.2.2 Temporal convolutional network

Causal convolutions characterize a temporal convolutional network (TCN). The vanilla temporal convolutional network operates with a causal constraint, i.e., the output y_n depends only on the inputs $x_1, \ldots, x_n$. Moreover, the model accepts a sequence in the input and produces a sequence of the same length in the output.

Causal convolution

The architecture leverages a 1D fully convolutional network [13], where each layer keeps the same shape as the input using zero padding. Next, causal convolutions ensure that the output at time t is dependent only on neurons $x_1, \ldots, x_t$ from the preceding layer. This basic design of a causal convolution requires an immensely deep neural network or unusually wide kernels [14]. These disadvantages can be solved by a residual connection which helps a particular TCN block to learn modifications [15].

Dilated convolution

A dilated convolution enables a model to use longer history. With a linear number of convolutional layers, the network can leverage an exponentially larger receptive field [14]. The receptive field can be controlled by the kernel size k and the dilation factor d. Specifically, the dilation factor $d = 1$ is used in a regular convolutional layer. An example of a causal convolution and a dilated convolution is shown in Figure 2.7.

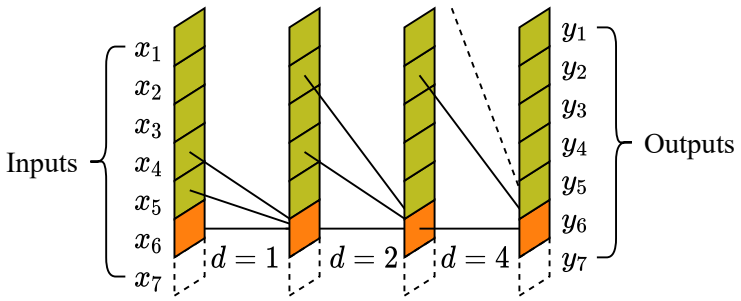


FIGURE 2.7: This temporal convolutional network has the input layer, two hidden layers, and the output layer. The dilated factor d exponentially grows with the number of layers. The kernel size of each layer is equal to three.

2.3 Self-attention mechanism

A self-attention mechanism [16] has recently become an essential block of every modern NLP model but also in the computer vision domain [17, 18]. The self-attention mechanism lets each input interact with each other and returns a matrix that denotes the importance of each input value.

Internally, the self-attention mechanism contains a key matrix, a value matrix, and a query matrix. These matrices update their weights during the training phase. Next, the score of each input, with respect to a particular input value, is computed by taking the dot product of a query vector and a corresponding key vector. Afterward, the scores are normalized by a constant value to have more numerically stable gradients. Then the softmax activation function is applied to the result, ensuring that all entries are non-negative and sum up to one. The softmax score is multiplied with each vector which creates weighted numerical vectors. The weighted vectors are summed up, which produces the self-attention score for a particular input. This process is repeated for all other inputs. The calculation of a self-attention score is depicted in Equation 2.3 where Q denotes the query matrix, K denotes the key matrix, V denotes the value matrix, and d corresponds to the dimension of the key vector.

The performance of the self-attention mechanism can be further improved by a multi-head attention. This allows us to create multiple triples of key, value, query matrices. Each triple can capture different information since all weights of these triples are randomly initialized [16].

$$attention = \texttt{softmax}(\frac{QK^T}{\sqrt{d}})V \tag{2.3}$$

2.3.1 Self-attention & convolutional layers

The self-attention mechanism can improve the performance of computer vision tasks, and thus self-attention models can compete and outperform convolutional models [19] while requiring fewer trainable parameters [20]. A convolutional neural network cannot capture the dependencies of features that are spatially distant. On the contrary, a model with self-attention layers can learn such global dependencies as it learns the relationship between each input to the others [21]. Specifically, a multi-head self-attention layer is just a generalized convolutional layer. The authors [22] showed that a self-attention layer could be equivalent to an arbitrary convolutional filter in a particular convolutional layer, assuming the usage of relative positional

encoding and enough heads. The relative positional encoding efficiently allows the self-attention mechanism to produce a different representation of each sequence element using the relative distance between the elements [23].

2.4 Autoencoder

An autoencoder is a neural network that learns an approximation of the identity function, i.e., it tries to copy the input to its output. An autoencoder usually reduces the input until a hidden layer h which describes a code (a compressed representation of the input). This part of an autoencoder is named an encoder. Next, the decoder reconstructs the code to the initial shape of the input [2]. Figure 2.8 shows an example of an autoencoder comprising fully connected layers.

A particular autoencoder is useful as long as it does not learn to precisely copy the input to the output. The autoencoder usually incorporates mechanisms that force it to learn only an approximation of the input. Therefore, an autoencoder can learn valuable properties of the underlying probability distribution of a data set.

An autoencoder does not require annotated data, and thus it approximates the identity function in an unsupervised manner. However, our task, anomaly detection in log files, needs annotated data since an autoencoder is supposed to learn the normal state of logs. The hypothesis assumes that a particular autoencoder trained on normal data examples could reconstruct anomalous data examples with a significantly higher loss. Therefore, our task requires autoencoders trained in a semi-supervised manner.

Besides the autoencoders, which use only fully connected layers, we propose various autoencoders which leverage convolutional layers and the self-attention mechanism. The detailed overview of proposed autoencoders is in Section 4.4.

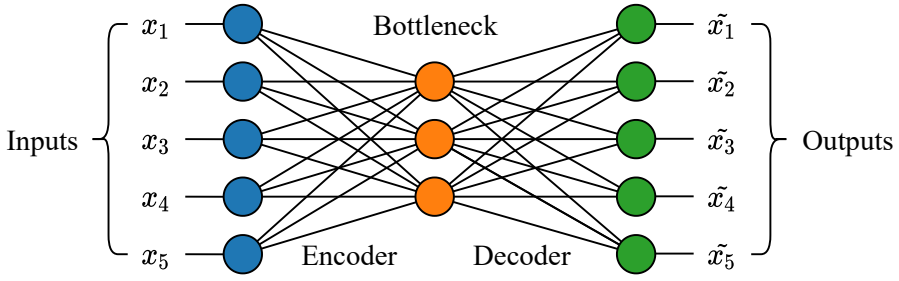


FIGURE 2.8: An example of an autoencoder that consists of an encoder, a bottleneck, and a decoder. The input is reduced until the bottleneck, which describes a code. The code is a compressed representation of the input. Finally, the decoder reconstructs the code to the input shape.

2.4.1 Regularized autoencoders

An autoencoder might directly copy the input to the output and thus achieves zero loss on the data set. This is not a desirable behavior of the autoencoder. Therefore, it is crucial to impose a regularization technique on the autoencoder [24]. One possible solution is to introduce a bottleneck, as is shown in Figure 2.8.

The first approach is known as sparse autoencoders. The autoencoder is forced to have only a small number of active hidden units simultaneously. The sparsity can be achieved by introducing an additional term to the loss function. The term

might be L1 regularization or the Kullback–Leibler divergence, which measures the dissimilarity of two probability distributions.

Another approach resides in denoising the input [25]. The input is corrupted by some noise at the beginning, such as white Gaussian noise. Next, the autoencoder tries to reconstruct the original undistorted input. The model has to extract informative features in order to learn the denoising accurately.

2.4.2 Applications of autoencoders

Autoencoders have been used for dimensionality reduction, while a classical approach is Principal Component Analysis (PCA) [26]. An autoencoder with only the linear activation function strongly relates to PCA. However, the main advantages of autoencoders reside in the usage of non-linear activation functions. These autoencoders may outperform PCA if the manifold is non-linear. On the contrary, PCA is optimal as a linear projection [24]. Specifically, we used autoencoders for dimensionality reduction in Subsection 4.4.9.

Another application of autoencoders is anomaly detection. An autoencoder can learn the latent space of normal data examples. Once the autoencoder is appropriately trained, it should reconstruct normal data points with a considerably lower loss. On the contrary, the anomalous data points should yield a higher reconstruction loss. Although there are various other applications of autoencoders, we mentioned dimensionality reduction and anomaly detection applications as they are interwoven with our main task.

Chapter 3

Related Work

We first define what anomalies are and how we can deal with them. Next, we provide an intensive overview of the current state-of-the-art methods used for anomaly detection and lastly aim our focus on anomaly detection in log files.

3.1 Anomaly detection

Anomalies are the data points dissimilar to the rest of the data based on appropriate metrics [27]. The rest of the data is called normal data. Anomaly detection is related to the problem of identifying patterns in a data set that significantly deviate from the expected behavior [28].

3.1.1 Categories of anomalies

We classify anomalies into three categories by their types akin to authors [27, 29, 30].

Point anomalies

We say that a particular data point is anomalous if it significantly deviates from the rest of the data points [27]. Point anomalies are the simplest type of anomalies, and an example might be a temperature of a human body. Let us assume that an average temperature of a human body is 36 °C for the sake of simplicity. We conduct five measurements and four times the temperature is 36 °C. During one of the measurements, the temperature reaches 40 °C, and thus it is a point anomaly. Figure 3.1 shows one such example.

Contextual anomalies

A particular data point belongs to this category if it is anomalous only in a specific context but not otherwise. There are contextual and behavioral attributes that define each data point. Contextual attributes determine the context or the neighborhood of the data point. In contrast, behavioral attributes define the noncontextual characteristics of the data point [27]. We often observe contextual anomalies in time-series data such as temperature, stocks, and logs. Figure 3.2 depicts an example of a contextual anomaly.

Collective anomalies

A collection of data points is a collective anomaly if it is anomalous with respect to the entire data set. Individual data points are not usually anomalies, but their occurrences together represent an example of a collective anomaly [27]. Figure 3.3 is an example of collective anomalies.

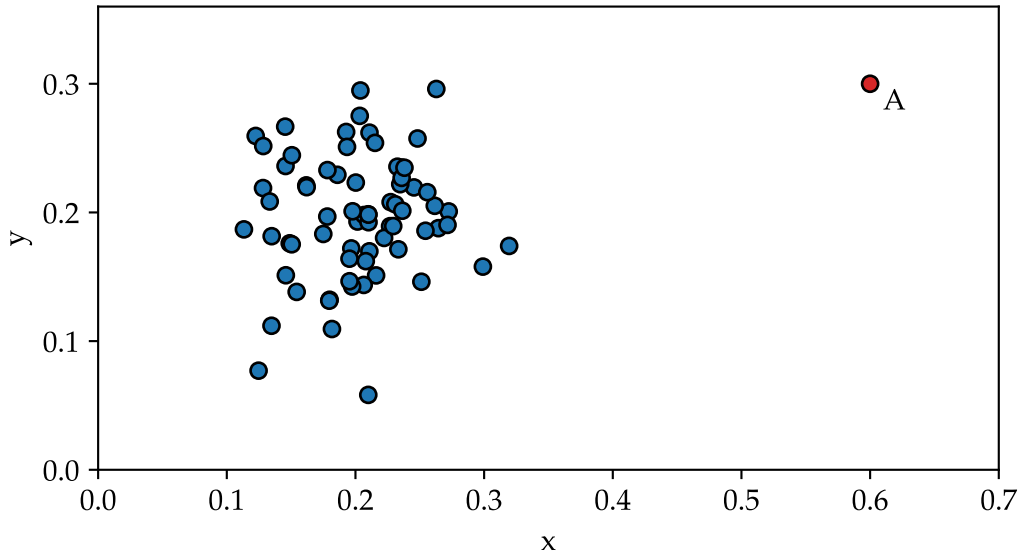


FIGURE 3.1: A point anomaly example. The blue data points represent normal data, whereas the data point A is labeled as an anomaly.

3.1.2 Presence of data labels

We distinguish three types of anomaly detection methods based on the data labels.

Supervised anomaly detection

All methods which belong to this category assume that there exists a label for each data point, i.e., it is known whether a particular data point is an anomaly or not. Support vector machines (SVM) [31], a logistic regression [32], and decision trees [32, 33] are representative examples of supervised anomaly detection methods. These methods usually outperform the following categories [32], but collecting labels may be costly and, in some cases, even unfeasible. Another issue is that the number of anomalous data points in a particular data set is notably fewer than normal data points. Supervised anomaly detection methods are not in the scope of this thesis.

Semi-supervised anomaly detection

Semi-supervised methods assume that all training data points belong to the normal class. Since these methods do not need anomalous data points in the training phase, they are widely applicable in the anomaly detection domain [27]. The authors [34] adapted SVM to one-class SVM. One-class SVM tries to estimate a function on normal data points, which is negative for anomalies and positive otherwise.

Our proposed methods, presented later in this thesis, belong to this category. Autoencoders are an exquisite example of semi-supervised methods, e.g., authors [35] developed a robust autoencoder to detect anomalies in images.

Unsupervised anomaly detection

Unsupervised methods do not require any training data points, and therefore they are also widely applicable in the anomaly detection domain. These methods usually assume that the number of anomalies in the data set is significantly lesser than

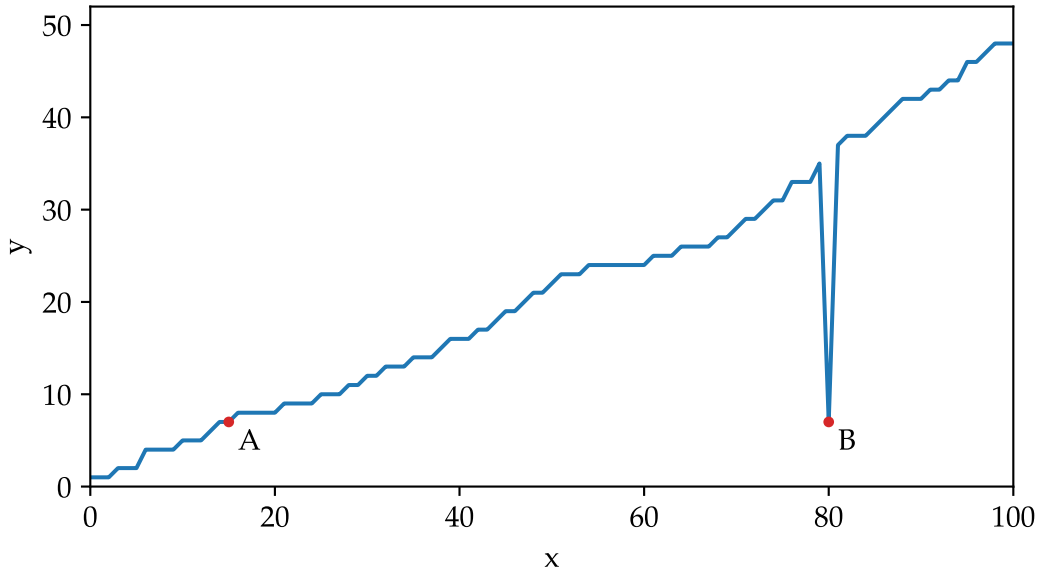


FIGURE 3.2: A contextual anomaly example. The value of the data point A might be expected within a range close to $x = 20$, but the same value (depicted as the data point B) is definitely anomalous with respect to its neighborhood.

the number of normal data points [27]. Specifically, the authors [36] proposed an unsupervised method based on the local outlier factor (LOF), which measures an anomaly score of each example. The anomaly score depends on how isolated a particular data point in space is with respect to the surrounding neighborhood. Another example may be the Isolation Trees algorithm, which was presented by the authors [37]. Partitioning of individual data points is repeated recursively until all data points are isolated. Anomalous data points have noticeable shorter paths to the root. These data points are likely to be separated in early partitioning because they have contrasting attribute values with respect to the rest of the data set [37].

Methods proposed in [36, 37] are used as the baseline methods in our experiments; see Subsection 4.4.2.

3.1.3 The output of anomaly detection methods

One crucial aspect of every anomaly detection method is the output form in which the anomalous behavior is reported. Typically, we can distinguish two output forms, namely an anomaly score and an anomaly label, according to [27, 38].

An anomaly score, usually expressed as a real number, describes the certainty of being an anomaly for each data point. Thus, the output is a ranked list of anomalies, and a domain-specific threshold is applied to identify anomalies. This output form provides us with more information than anomaly labels.

An anomaly label, usually expressed as a binary label, assigns to each data point either a normal or an anomalous label. This output form does not allow us to use any domain-specific threshold. However, the threshold might be controlled indirectly through parameter tuning in a particular anomaly detection method.

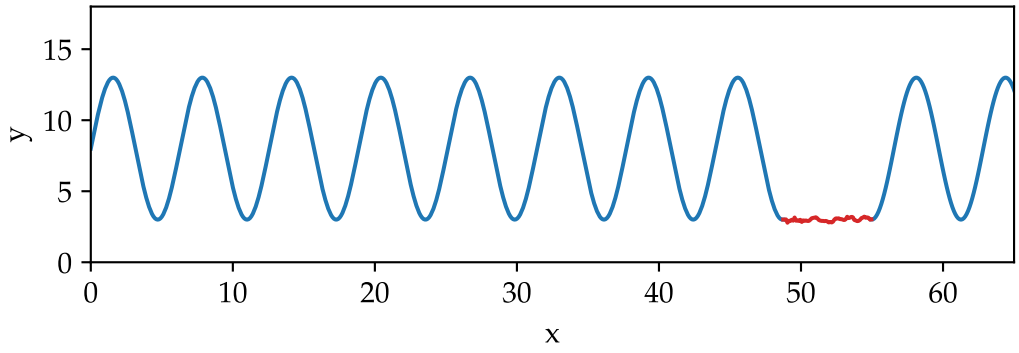


FIGURE 3.3: A collective anomaly example. The red color denotes an anomaly because the range contains very similar values for a longer period of time than it was expected.

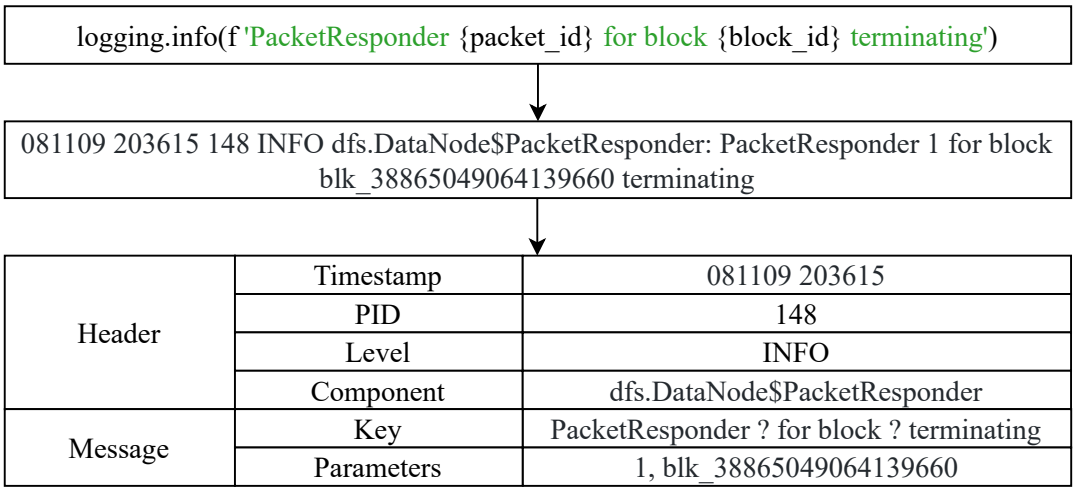
3.2 Anomaly detection for log files

Until now, we have described general-purpose methods for anomaly detection. We aim our focus on methods used to detect anomalous data points in log files and emphasize how essential logs might be nowadays.

3.2.1 Importance of logs in modern large-scale systems

With the current demand for cloud services, it has become increasingly important to provide highly available and error-free services. Otherwise, it would cause considerable hardship to both providers and users. Therefore, companies invest much money to forestall outages of their services [39]. High availability of services can be achieved by quickly identifying the root cause, and generating logs is one possible solution. Therefore, every application and every process generates millions of log messages every day [39], informing an administrator about performance, failure, and availability of the services. Since data centers consist of hundreds of running components on thousands of nodes, sometimes even distributed on different continents, the importance of automated log processing arises when troubleshooting and diagnosing operational problems.

Logs are valuable resources of information for debugging purposes in the early phase of development or monitoring deployed applications. Since a human being is the main recipient, logs usually contain information in a human-readable format such as plain text. The common structure is depicted on Figure 3.4. Although there is no universally accepted terminology, we describe the most all-important terms. It is worth noticing that some of these terms are used interchangeably. A log line is also known as a log statement in the research community. The log statement usually corresponds to a single line of text in a particular log file. The log statement comprises two parts: a log header and a log message. The log header records information about a timestamp, a level of severity, or a process identification number (PID). Since the same formatting is used for all types of logs, they are relatively easy to parse. The log message is further divided into a log key and log parameters. The log key is a static part that is the same for all log lines corresponding to a particular event. On the other hand, the log parameters are variables that may vary within one event. A representative example of log parameters is an IP address or the size of a file.



Header	Timestamp	081109 203615
	PID	148
	Level	INFO
	Component	dfs.DataNode$PacketResponder
Message	Key	PacketResponder ? for block ? terminating
	Parameters	1, blk_38865049064139660

FIGURE 3.4: An example of the process from creating a log line in the programming language Python[1] to proper log parsing.

3.2.2 Traditional approaches for log processing

Traditionally, a log anomaly detection approach comprises log collection, log parsing, feature extraction, and anomaly detection. This general framework was first introduced in [32].

Log collection

Collecting log files and providing them on demand is a crucial feature of every large-scale system. Less complex systems usually use only simple log files. However, the need for a more sophisticated tool arises as the systems grow. We can mention syslog[2] or Google Cloud Logging[3], but for this thesis, we used publicly available data sets, and thus log collection is not assumed in this thesis.

Log parsing

As mentioned above, log files contain semi-structured data, which should help administrators with a root cause analysis. The goal of log parsing is to transform semi-structured data, i.e., log files, into non-specific structured data. A very comprehensive overview of the performance of different log parsers can be found in [40]. Even though the state-of-the-art log parsers can achieve high accuracy, it is worth mentioning that the log parsing task is challenging and none of the log parsers is faultless.

The log files usually contain log statements from different tasks which might be written concurrently, e.g., in a distributed file system. Fortunately, the individual log statements often provide a unique task identification number which eases the belonging to a group. A particular log key can appear in more than one task [41]. Therefore, it is essential to process the intertwined log statements carefully.

The oldest approach is based on handcrafted regular expressions, which extract log keys and log parameters [42].

The second approach, the source code analysis, was proposed in [43]. The authors tried to parse log lines more accurately using inferring the structure from print

[1] `https://www.python.org`
[2] `https://tools.ietf.org/html/rfc5424`
[3] `https://cloud.google.com/logging`

statements. Object-oriented programming causes several limitations, such as overwriting the `toString` method. Moreover, the source code may not be available at all when using proprietary libraries.

Finally, several data-driven approaches were proposed, such as hierarchical clustering [44], data clustering and line pattern mining [45], and a fixed depth parse tree [46]. We use the newer version of Drain [46], namely Drain3[4], as the baseline approach for parsing logs in this thesis.

Feature extraction

Extracting the most appropriate but still uncorrelated features is an essential subtask in every machine learning project. We can let a model decide itself [47, 48] which features are indispensable for making a prediction. We can also hardcode the features using an empirical study.

Since we are dealing with semi-structured data, namely log files, the feature extraction is often related to NLP techniques. The basic idea is to use the bag-of-words technique [49]. This technique creates a numerical feature vector where each entry represents the number of occurrences of a particular word, optionally normalized by a weighting method such as TF-IDF [50]. We can tailor this approach to log files where each entry in a vector means the number of occurrences of a particular log key [32]. Figure 3.5 depicts one such example. The log keys are estimated on a training data set, but a testing data set might also contain different log keys. The easiest solution is to skip those log keys that are not present in the training data set. Other possible solutions were proposed in [51], but retraining a model periodically as new log keys appear is still the most recommended solution.

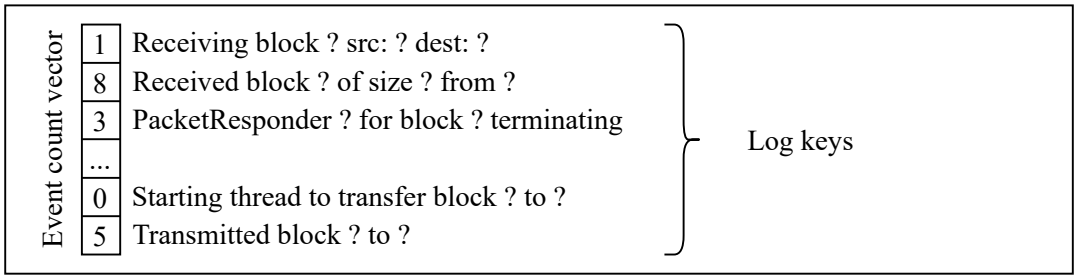


FIGURE 3.5: This example describes how a numerical representation of log lines belonging to a particular identification number can be constructed. Each entry in the vector is associated with a unique log key. The occurrences of individual log keys are calculated and assigned to corresponding entries of the vector.

A better approach seems to be learning embedding vectors. An embedding vector is a dense, fixed-size numerical vector built using co-occurrence statistics and their distributional hypothesis [50]. The best-known models for computing word representations in an unsupervised manner are Skip-gram and CBOW (continuous-bag-of-words). The Skip-gram model learns to predict a word embedding vector using the surrounding neighborhood. On the contrary, the CBOW model predicts a word embedding vector using its context. The context consists of a bag of words contained in a fixed size window in the word's neighborhood [52]. The Skip-gram model is used in this thesis for log representations as numerical vectors.

[4] `https://github.com/IBM/Drain3`

Authors [51] leveraged representing log lines as embedding vectors as they proposed a simple but effective method, which captures both syntax and semantic information from log keys. The semantic and contextual information also extracts the method [53] which detects anomalies by utilizing a Long Short-Term Memory (LSTM) model [54]. The most general method [41] utilizes the whole log, including a timestamp, a PID but also log parameters. Next, the deep learning model (LSTM) and classic mining approach detect anomalies in log files. Akin to the method [41], our proposed methods leverage whole logs, i.e., including log parameters, while learning the embedding vectors.

Although the logs contain textual information, the language is constrained, and thus we cannot use a pre-trained language model, which would perform poorly.

Anomaly detection

In Section 3.1, we described methods that can be successfully applied to anomaly detection. Here, we specifically focus on anomaly detection for log files. The methods can be classified into three categories based on the availability of data labels, similarly to Subsection 3.1.2.

The first category describes unsupervised methods. Traditionally, PCA was used for anomaly detection to detect run-time system problems automatically. To our best knowledge, authors [43] were the first who successfully applied PCA on log files. They constructed two subspaces using PCA and calculated a projection distance between the space and a feature vector, which consists of event occurrences. If the distance is greater than a predefined threshold, the feature vector is classified as an anomaly. The authors [55] proposed a method that firstly measures the cosine similarity and then performs log clustering based on the agglomerative hierarchical clustering technique.

The supervised methods belong to the second category. Logistic regression [32] is first trained on a training data set containing both normal and anomalous data points. Next, in the testing phase, it returns the probability of being anomalous for each queried data point. If the value of the data point is greater than 0.5, the example is classified as anomalous. Decision trees [33] were used to diagnose failures on the Internet websites. An example traverses through an individual path from the root to a leaf during the testing phase. Each leaf of a decision tree is annotated with either the anomalous or the normal label. Once the example reaches a leaf, the leaf returns the corresponding label as a prediction. The most mature supervised approaches leverage LSTM [41] and convolutional neural networks (CNN) [56].

Supervised methods beat easily unsupervised methods but at the cost of log annotation, which often requires domain experts. Therefore, we focus mainly on semi-supervised methods in this thesis, which are in-between supervised and unsupervised methods. As discussed earlier, semi-supervised methods need only non-contaminated data, i.e., log files without anomalous logs. Although there are semi-supervised anomaly detection methods, it is not usual to apply them on log files.

Chapter 4

Solution Approach

In the previous chapter, we discussed the state-of-the-art methods for anomaly detection and anomaly detection in log files.

Traditional anomaly detection methods for log files need extracted log keys to create a numerical representation on top of which a machine learning model is built. However, this approach brings several limitations.

Firstly, preprocessing techniques usually suffer from errors caused by imperfect parsing methods. These parsing methods have their limitations in the log key detection.

Another issue arises after the log key detection. Ideally, we must know all possible log keys in advance, which is not always feasible. Periodically retraining a model is proposed as an alternative solution as the new log keys appear.

The last obvious limitation is that we lose much information in the preprocessing phase since only extracted log keys are used for creating a feature vector. We could manually incorporate new features such as a timestamp, a PID, a level of severity, or even log parameters. It was shown in [41, 51] that adding this information can improve the performance of anomaly detection systems.

Many state-of-the-art anomaly detection methods rely on annotations, as it was mentioned in Chapter 3. Together with the limitations mentioned above, we identify the bottlenecks of modern anomaly detection systems for log files. In this chapter, we propose multiple methods which deal with the mentioned bottlenecks. Furthermore, we introduce a data set which the proposed methods use and describe training, validation, and testing settings.

4.1 From HDFS to creating a data set containing logs

Hadoop[1] is a framework containing many open-source software components which ease solving problems involving a massive amount of data. One of these components is the Hadoop Distributed File System (HDFS). HDFS is a distributed file system that can run on commodity hardware. HDFS is highly fault-tolerant, provides high aggregate bandwidth, and is suitable for applications handling large data sets [57].

The HDFS data set[2] is a collection of logs from a Hadoop cluster running on more than 200 nodes. Console logs are written directly to the storage on each node at runtime. Next, the logs are aggregated to a single file offline. The level of severity is set to the default HDFS logging level. The logs are grouped by a particular block ID, and each collection of logs is annotated by the normal or the anomalous

[1] `https://hadoop.apache.org`

[2] `https://zenodo.org/record/3227177/files/HDFS_1.tar.gz?download=1`

```
...
081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 1
    ↪ for block blk_-1608999687919862906 terminating
081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 2
    ↪ for block blk_-1608999687919862906 terminating
081109 203519 145 INFO dfs.DataNode$PacketResponder: Received block
    ↪ blk_-1608999687919862906 of size 91178 from /10.250.10.6
...
```

FIGURE 4.1: A few log lines from the HDFS data set. Each line consists of a timestamp, a PID, a level of severity, a log component, and a log message, respectively.

label. The labels have been discussed and further confirmed by Hadoop developers [43]. Figure 4.1 depicts a few lines of the original HDFS data set before applying any changes. An individual log line consists of a header (a timestamp, PID, a level of severity, and a component) and a message (a log key and log parameters); see Figure 3.4 for more details. The data set contains 11,175,629 lines in total. Each line belongs to a particular block according to its `blk_$ID`, where `$ID` denotes a unique integer. The data set was collected within 38.68 hours, i.e., the first log was recorded on 9 November 2008 in the evening and the last log on 11 November 2008 at noon. The total number of blocks is 575,061, where 16,838 blocks are anomalous, which translates into 2.93% of anomalous blocks in the whole data set. More statistics about the data set are listed in Table 4.1. Finally, Figure 4.2 depicts the block size distribution, as not all blocks contain the same number of log lines. The most common number of log lines within a particular block is 19, followed by 13 and 25. Each block contains between 2 and 298 log lines.

There also exists a larger HDFS data set containing over 71 million log lines. Unfortunately, we could not use this data set in our work as it comprises only unlabeled data.

HDFS data set statistics	
Total size	1.58 GB
Labels	per block ID
Time span	38.68 hours
Maximum time span within a block	15 hours
Number of log lines	11,175,629
Number of unique log keys	48
Number of total blocks	575,061
Number of anomalous blocks	16,838
Relative number of anomalies	2.93%

TABLE 4.1: Detailed statistics of the HDFS data set.

When building a machine learning project, the general recommendation requires splitting a data set into training and testing parts. The HDFS data set is split according to individual blocks, i.e., log lines belonging to a particular block are a part of either the training data set or the testing data set. Since we need also tune hyperparameters (see Section 4.3), an additional validation part is necessary. Therefore, the HDFS data set is first split into training and testing data sets in ratio $10 : 1$. It

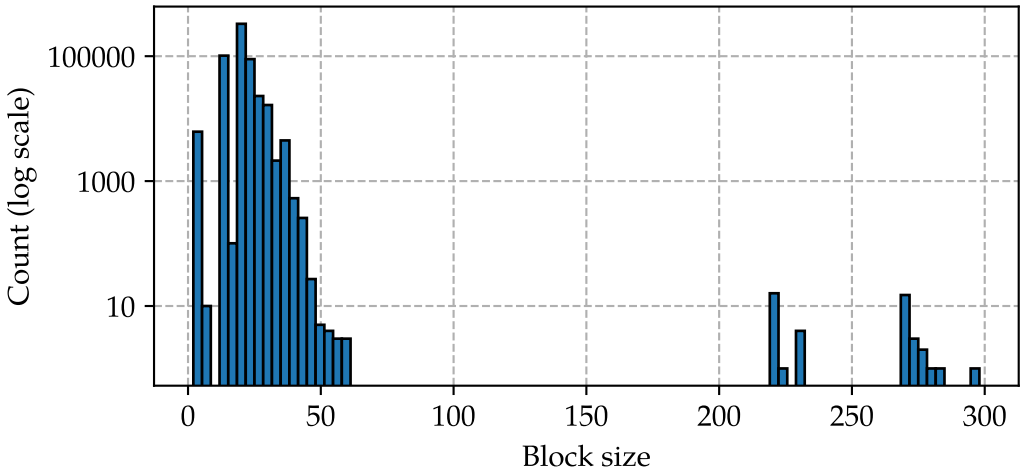


FIGURE 4.2: The histogram clearly shows that most of the blocks have the number of log lines strictly less than 100. Although several blocks contain more log lines, the number of such occurrences is relatively small with respect to the whole distribution.

is a good practice not to touch the testing data during the development process. Finally, the training data set is divided into the training data set and the validation data set. The training and validation data sets follow the exact split ratio (10 : 1).

4.1.1 On challenges of imbalanced data sets

The majority of machine learning algorithms require a similar number of occurrences across all classes. However, the distribution of a particular data set might be skewed. Therefore, the machine learning algorithm is biased towards the majority class. Unfortunately, the minority class is the one, which is crucial in many real-life applications, such as fraud detection, histopathologic cancer detection, and anomaly detection in logs [58].

Researchers distinguish three main approaches tackling imbalanced data sets, namely a data-level approach, an algorithm-level approach, and a combination of two previously mentioned approaches.

Data-level methods try to modify the original data set in a suitable way for an arbitrary machine learning algorithm. We usually have two choices, either to oversample the minority class or to undersample the majority class. However, these approaches often introduce meaningless data examples and remove essential data examples, respectively [58].

Algorithm-level methods try to modify the machine learning algorithms so that they can adequately discriminate the classes, and thus they are not biased towards one of the classes [58]. The solution is often based on penalizing the samples from the majority class [59]. However, there arises a problem to set the cost of the individual classes precisely.

The last approach combines two previously mentioned approaches in order to leverage their strengths and reduce their weaknesses [58].

Circumspection while using an imbalanced data set is crucial, especially for supervised machine learning. The recent research [60] showed that semi-supervised learning is robust to imbalanced data sets. Therefore, we do not modify in any way

the HDFS data set, i.e., oversampling the anomalous class or undersampling the normal class, since we leverage semi-supervised learning.

Metrics

As the HDFS data set is imbalanced, we must carefully select appropriate metrics. We cannot use accuracy even though this metric is the recommended choice for many real-life applications [61]. The minority class has little impact on the overall accuracy compared to the majority class.

Let us assume the following example. The data set contains 100 examples where 98 examples belong to the normal class, and two examples belong to the anomalous class. In such a scenario, the accuracy can be 98% by predicting the normal class every time. Unfortunately, this model is useless as we want to detect the rare events, i.e., the anomalous class in our example. We rather sacrifice accuracy for predicting the minority class correctly.

We define several well-suited metrics for the HDFS data set and are used in anomaly detection for log files in the research community [32, 42].

The precision (Equation 4.1) informs us how many of the predicted anomalous data points are truly anomalous. On the other hand, the recall (Equation 4.2) informs us how many of all anomalous data points in the testing data set were identified as anomalous. In general, there is a trade-off between the precision and the recall, i.e., maximizing one of the metrics often means decreasing the other metric. Fortunately, there exists a metric that takes into consideration both mentioned metrics. The metric is the harmonic mean of the precision and the recall, and it is called the F1-score (Equation 4.3). The F1-score is the primary metric in all our experiments.

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{4.1}$$

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{4.2}$$

$$F1\text{-}score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{4.3}$$

4.2 Creating embeddings from semi-structured data

Once the HDFS data set is grouped by `blk_$ID`, as depicted in Figure 4.3, we need to convert the actual blocks with logs to a numerical representation. We introduce several approaches used in this thesis.

4.2.1 Baseline approach

In the beginning, we researched state-of-the-art unsupervised methods used for anomaly detection in log files. These methods are delineated in Subsection 4.4.2. It is sufficient for now that these methods require a single numerical vector as the input. Furthermore, we could not fully leverage all log lines, as there was a need to aggregate the log lines into a single feature vector.

We created a bag-of-words representation. Each entry corresponds to a particular log key and incorporates the number of occurrences of the given log key

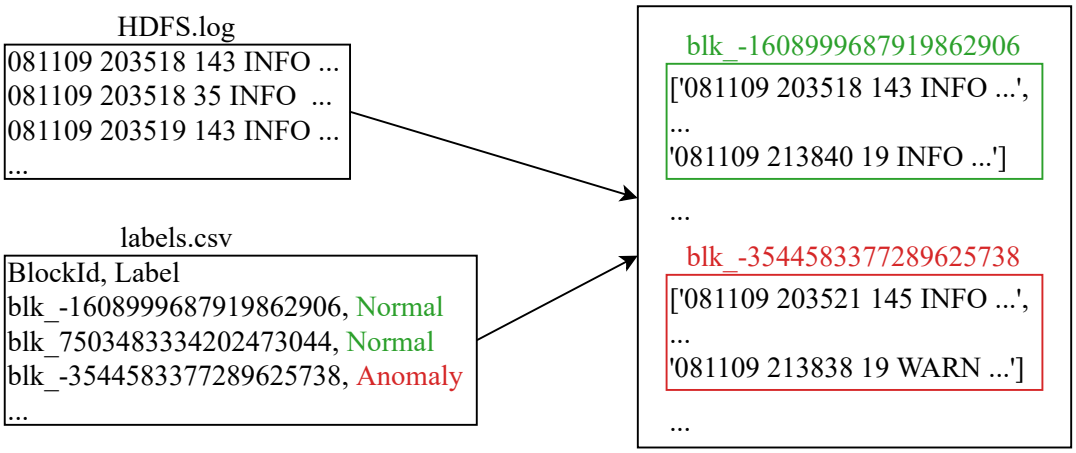


FIGURE 4.3: This is an actual snippet of the HDFS data set. The log lines and corresponding labels are loaded into memory. Next, each line is assigned to one block (bucket) according to its `blk_$ID`. Finally, the whole block is annotated with either the normal or the anomalous label.

within a block; recall Figure 3.5. The obtained vectors can be centered by subtracting the mean vector and normalized by the weighting method TF-IDF. We applied both preprocessing techniques to the baseline methods.

4.2.2 Representation learning with fastText

The proposed models are capable of handling a matrix as the input. Therefore, no additional aggregation is needed, and it is sufficient to convert each log line into a numerical vector. The authors [42] have stated that embeddings created by an unsupervised fastText[3] model perform surprisingly well on semi-structured data, such as log files. The actual settings of the model were adopted from [42], i.e., the dimension of embeddings is 100, and the model is trained using the character n-grams from 3 to 6. We trained the model only on the training part of the HDFS data set to prevent overfitting. The process of transforming log lines into embeddings is depicted in Figure 4.4.

A particular log line is assumed to be a sentence that is essential from the fastText point of view. The actual calculation of a sentence vector differs from averaged word vectors. Firstly, each word vector is divided by its Euclidean norm. Then, the sum of all normalized word vectors is divided by the number of word vectors whose Euclidean norm is non-zero.

Besides learning the embeddings, we implemented an additional feature. The feature keeps the information about time deltas between a pair of succeeding log lines. The implementation of the feature is delineated in Subsection 4.5.3.

Cropping blocks in the HDFS data set

The log representations introduced in Subsection 4.2.2 preserve the same number of rows as the initial block had. Recall Figure 4.2 which shows the distribution of block sizes. All the models proposed in Section 4.4 need input in the form of matrices with the same shape. Therefore, additional padding is required in order to fulfill

[3] `https://fasttext.cc`

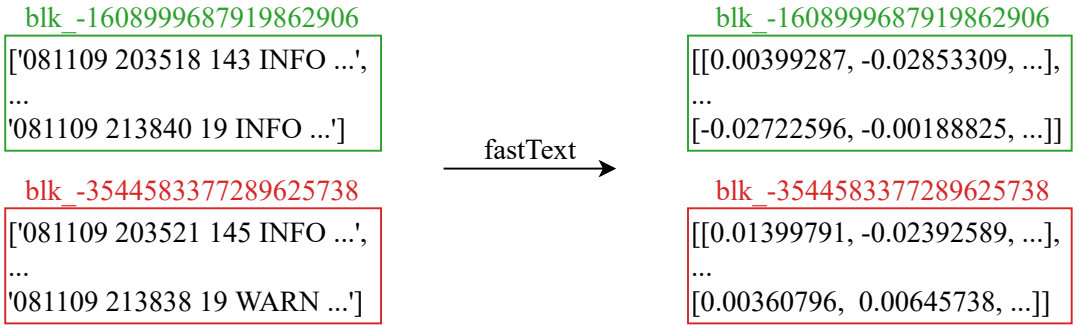


FIGURE 4.4: FastText creates a numerical vector for each log line in all blocks, i.e., initially, a block contained a list of log lines, then the block contains a list of numerical vectors. We used a trained fastText model to predict the values depicted in this figure.

the desired input shape. We leave the block size as a model hyperparameter. The actual implementation is delineated in Subsection 4.5.3.

4.3 Hyperparameters

A hyperparameter is a parameter whose value is set before the training phase. The value is unchangeable during training a model. The hyperparameters may influence both the training time and the final performance of a particular model [9].

Neural networks generally have a large number of hyperparameters. Although we use various models in Section 4.4, we describe the vital hyperparameters below.

The learning rate is a parameter of an optimization algorithm which determines the magnitude of change in weights to minimize a loss function. The batch size specifies the number of examples used for one update of weights. The topology, the size of a neural network, and the block size are complex hyperparameters since their search space is enormous. Surprisingly, we assume that the number of epochs is also a hyperparameter of proposed neural networks as we train the models in a semi-supervised manner. Moreover, we cannot easily find the point where a particular model starts overfitting on the training data set. This is caused by the nature of semi-supervised machine learning as there is no validation data set which tracks the loss on the validation data set simultaneously. Next, the dropout refers to skipping some neurons during the training phase. This hyperparameter controls the percentage of skipped neurons. The neurons are chosen at random. Last but not least, an activation function and the kernel size in a convolutional layer are considered as additional hyperparameters.

Besides a training data set, the hyperparameter tuning phase also requires a validation data set. The validation data set contains examples that have not been exposed during the training phase. The loss on the validation data set appraises whether the hyperparameters were set appropriately. Furthermore, k-fold cross-validation provides more precise statistics but at the cost of a noticeable deceleration of the whole training and validation process [9].

There exist several algorithms for hyperparameter tuning [2]. A grid search is a naive approach which evaluates only combinations of hyperparameters given beforehand. A random search [62] enables an extensive search in a search space of hyperparameters. It is still a preferred choice over Bayesian optimization, as mentioned in the course [7].

4.4 Proposed models

As we know all the details about the data set, we may propose machine learning models which leverage the embeddings introduced in Subsection 4.2.2. We start by describing the choice of a loss function and an optimizer.

4.4.1 The choice of a loss function and an optimizer

The vast majority of proposed models are based on a neural network, and thus the backpropagation algorithm is used. The models try to learn an approximation to the identity function. The most appropriate loss function, which we also selected, is usually the mean squared error; see Equation 4.4. Nevertheless, this loss function might be prone to outliers [9].

There is a large pool of optimizers such as SGD, RMSProp, and Adam. We decided to use the Adam optimizer as it is a recommended choice [9, 42].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(ground\_truth_i - predicted\_value_i)^2 \tag{4.4}$$

4.4.2 Baseline models

The baseline models leverage the approach mentioned in Subsection 4.2.1. The input is a standalone vector which aggregates the information from all log lines belonging to a particular block. The obtained vectors can be centered by subtracting the mean vector and normalized by the weighting method TF-IDF. The training data set contains 48 different log keys, hence the input of the baseline models is a one-dimensional vector with 48 real numbers.

The following models are examples of unsupervised machine learning.

Local Outlier Factor model

The simplest baseline model is LOF (mentioned in Subsection 3.1.2), which measures an anomaly score of each example. We used the original unsupervised method proposed in [36]. However, there exists a modified semi-supervised method[4], which is trained on a data set without anomalous data examples.

The unsupervised method has two deciding hyperparameters, namely the number of neighbors and a distance metric. The number of neighbors is used in querying k-nearest neighbors, and the metric is used to calculate the distance between each k-nearest neighbor and the source data point.

A noticeable disadvantage of using LOF is the high computational complexity.

Isolation Trees model

The more sophisticated baseline model is Isolation Trees (mentioned in Subsection 3.1.2), where anomalous data points usually have shorter paths from the root to a particular leaf.

The essential hyperparameters are the number of trees, the number of examples drawn from a data set to train a tree, and the number of features drawn from all available features to train a tree. We leverage the bootstrap sampling. The bootstrap

[4] `https://github.com/scikit-learn/scikit-learn/blob/95119c13a/sklearn/neighbors/_lof.py#L19`

sampling is a technique where individual trees are trained on a data subset sampled from the source data set with replacement.

A grid search was used for tuning hyperparameters as Isolation Trees, and LOF models have a small number of hyperparameters.

Vanilla autoencoder

The last baseline model is a vanilla autoencoder which was originally created as a proof of concept. The crucial difference between this autoencoder and the other autoencoder-based models mentioned later on is the input. The input is only a one-dimensional vector per block akin to Isolation Trees and LOF models. Therefore, the autoencoder comprises only fully connected layers.

The proposed autoencoder and the other models introduced later in this section have a large number of hyperparameters. Therefore, a random search is used for tuning hyperparameters henceforth.

We tuned several hyperparameters, namely the number of epochs, the learning rate, the batch size, the topology and the size of a neural network, and the dropout.

4.4.3 TCN model

This model is based on the concept of TCN described in Subsection 2.2.2. TCN can process a sequence of an arbitrary length and outputs a new sequence with the same length as the input. A key characteristic is that the output at index i can leverage entries of an input sequence which occur before i (causal convolutions). Nevertheless, we annulled this limitation as our model should decide on an output using the whole block. Unfortunately, it brings a new limitation that only an odd kernel size greater than one can be used. The kernel size is used for calculating the padding size, and the result has to be an integer.

We created several modifications which contain TCN blocks, such as the model mentioned in Subsection 4.4.6. We build the architecture of a model which leverages the TCN block as an encoder and a decoder. The encoder and the decoder are linked using a bottleneck which consists of a fully connected layer. We call this model AETCN.

We tuned several hyperparameters using a random search, namely the number of epochs, the learning rate, the batch size, the topology and the size of a neural network, the dropout, and the block size. The list of hyperparameters is enriched with the number of neurons in the bottleneck of the AETCN model.

4.4.4 CNN1D model

The CNN1D model comprises two main blocks, an encoder and a decoder, respectively. The encoder is built upon 1D convolutional blocks. The block repeats the 1D convolutional layer followed by the activation function ReLU and the max-pooling layer. A particular 1D kernel convolves over a block, and an embedding dimension is seen as multiple feature maps. The decoder consists of several 1D transposed convolutional blocks. Each block repeats the 1D transposed convolutional layer, the activation function ReLU and an upsampling layer using the nearest neighbor algorithm[5].

[5] `https://pytorch.org/docs/stable/generated/torch.nn.Upsample.html`

Furthermore, we proposed a modified CNN1D model. The modified model adds a fully connected layer between the encoder and the decoder. This model is called AECNN1D.

The list of hyperparameters is akin to the one mentioned in Subsection 4.4.3. Moreover, we distinguish between the kernel size of the encoder and the kernel size of the decoder. Akin to the AETCN model, the AECNN1D model has an additional hyperparameter — a number of neurons in the bottleneck.

4.4.5 CNN2D model

The CNN2D model works similarly to the CNN1D model from a high-level overview. The major difference is that both the encoder and the decoder use the 2D convolutional layer and the 2D transposed convolutional layer, respectively. In contrast to the CNN1D model, the CNN2D model considers the input as a matrix with one channel. Therefore, a particular 2D kernel convolves over log lines as well as an embedding dimension.

We did not implement any modified models as the first results of these experiments were not promising.

The list of hyperparameters is the same as the one mentioned in Subsection 4.4.4.

4.4.6 CNN1DTCN model

This model combines the advantages of both 1D convolutional blocks and TCN blocks. The idea behind this model is as follows. Firstly, a few 1D convolutional blocks are applied to detect local dependencies and reduce the input dimension. Next, TCN blocks detect global dependencies and propagate them carefully through the model. Finally, a decoder (a block of the 1D transposed convolutional layer, the activation function ReLU, and an upsampling layer) reconstructs the output of the last TCN block back to the same shape as the input.

We tuned the following list of hyperparameters, i.e., the number of epochs, the learning rate, the batch size, the topology and the size of a neural network, the block size, the dropout, the kernel size of the 1D convolutional blocks, the kernel size of the TCN blocks and the kernel size of the 1D transposed convolutional blocks.

4.4.7 SACNN1D model

Thanks to the recent outstanding results of transformers[16, 63], we decided to try incorporating the self-attention mechanism into an autoencoder-based model. The implementation of the self-attention mechanism is straightforward since we can directly reuse the one mentioned in [16].

The proposed model combines 1D convolutional blocks, self-attention blocks, and 1D transposed convolutional blocks. Precisely, every pair of 1D convolutional layers is followed by the multi-head self-attention layer. The authors [64] recently recommended not to use an activation function straight before the self-attention layer. We follow this recommendation in this architecture and the architecture introduced in Subsection 4.4.8. A particular architecture of the SACCN1D model is designed so that the encoder contains exactly one self-attention layer. On the contrary, the decoder might contain at most one self-attention layer.

The list of hyperparameters comprises the number of epochs, the learning rate, the batch size, the topology and the size of a neural network, the kernel size of the 1D convolutional layers, the kernel size of the 1D transposed convolutional layers,

the dropout, the block size and the number of heads in the first and, optionally, in the second self-attention layer.

4.4.8 SACNN2D model

We tried to incorporate the self-attention mechanism into a model using 2D convolutional layers. Unfortunately, the implementation is not straightforward as in Subsection 4.4.7. We leveraged the ideas proposed in [18, 64]. In contrast to the SACNN1D model, we do not include the self-attention mechanism to an encoder or a decoder. However, we assume it as a bottleneck since the block contains multiple convolutional layers. Each model contains an encoder, a decoder, and the self-attention bottleneck. We reuse the implementation of the encoder and the decoder from Subsection 4.4.5. Figure 4.5 depicts the architecture of the bottleneck.

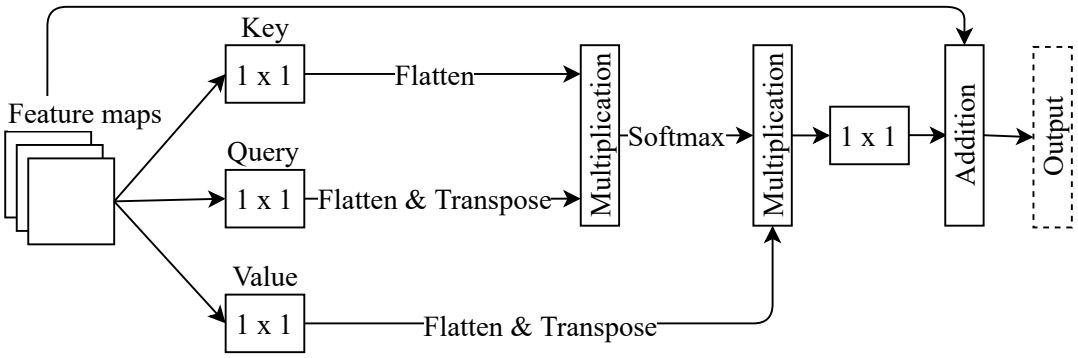


FIGURE 4.5: This 2D self-attention mechanism leverages 1D convolutional layers (denoted as 1×1). These convolutional layers enable us to change the number of feature maps within a self-attention block. Nevertheless, the final 1D convolutional layer produces the output with the same shape at the input. This architecture was inspired by authors [18].

The list of hyperparameters contains the number of epochs, the learning rate, the batch size, the topology and the size of a neural network, the kernel size of the 1D convolutional layers, the kernel size of the 1D transposed convolutional layers, the number of feature maps used in the bottleneck, and the block size.

4.4.9 Hybrid models

The last two proposed models leverage the AETCN model and its bottleneck, in particular. The bottleneck is viewed as newly compressed embeddings. Since the bottleneck is a one-dimensional vector, we can use the baseline models, namely the Isolation Trees model and the vanilla autoencoder. Figure 4.6 depicts the general architecture of a hybrid model.

Isolation Trees and AETCN

The Isolation Trees algorithm is used on top of the bottleneck of the AETCN model. The bottleneck consists of more neurons (vector entries) than the original input used in Subsection 4.4.2. Therefore, we hope that the new embeddings contain more information than the original input.

It is worth mentioning that we combine a semi-supervised model with an unsupervised model.

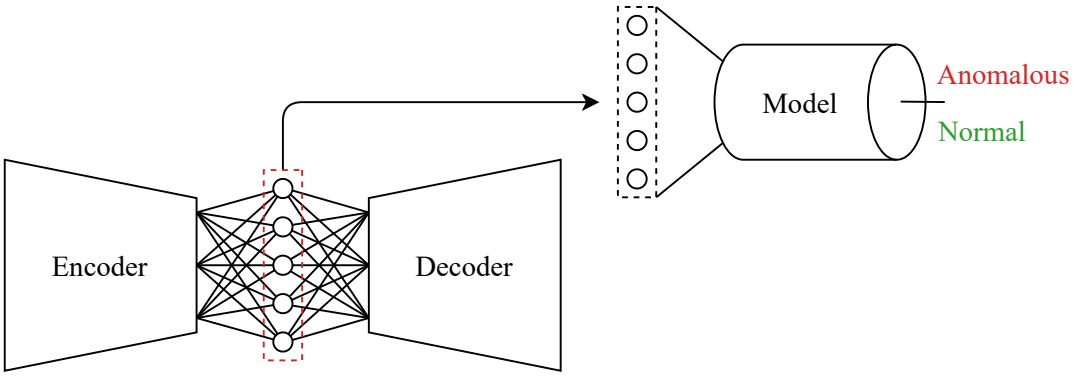


FIGURE 4.6: A hybrid architecture leverages the bottleneck of the AETCN model. The bottleneck is used as an input into another model, which makes the final predictions.

Vanilla autoencoder and AETCN

Another fully connected autoencoder is used on top of the bottleneck of the AETCN model. The autoencoder comprises only fully connected layers. The idea behind this hybrid model is akin to the previous hybrid model — we hope that we can further improve the performance on the HDFS data set.

We combine two semi-supervised models, autoencoders in particular. In the research community, this technique is sometimes called a stacked autoencoder [65].

4.5 Implementation

This section is divided into subsections which were implemented individually. Subsection 4.5.2 is dedicated to data preprocessing and data splitting. Subsection 4.5.3 regarding embedding learning follows together with creating an additional feature and cropping the blocks. Next, a model prototype is delineated in Subsection 4.5.4. Logging experiment results and a description of the RCI Cluster concludes this section.

4.5.1 Project overview

The whole project is written in the programming language Python 3.7. The reasons for choosing Python over other programming languages are platform independence, consistency, simplicity, a massive number of machine learning libraries and frameworks, and a wide community. The models are implemented using scikit-learn[6] and PyTorch[7] libraries. The scikit-learn library provides simple and efficient tools for machine learning tasks. The PyTorch library contains building blocks easing prototyping neural networks from early stages in research to production deployment. Last but not least, we leverage the NumPy[8] library, which brings well-optimized compiled C code into Python.

We decided to use the Cookiecutter Data Science[9] template to ease the collaboration with other researchers. The Cookiecutter Data Science template provides

[6] `https://scikit-learn.org/stable/`
[7] `https://pytorch.org`
[8] `https://numpy.org`
[9] `https://drivendata.github.io/cookiecutter-data-science/`

```
$ python prepare_hdfs.py --help
usage: prepare_hdfs.py [-h] [-in PATH/TO/FOLDER] [-out PATH/TO/FOLDER]
    ↪ n_folds

Process HDFS1 data set and split data into training, validation, and
    ↪ testing sets.

positional arguments:
  n_folds               a number of cross-validation splits

optional arguments:
  -h, --help            show this help message and exit
  -in PATH/TO/FOLDER    a location with HDFS1 data (HDFS.log and
     ↪ anomaly_label.csv) (default: ../../data/raw/HDFS1)
  -out PATH/TO/FOLDER   a location where all intermediate data will be
     ↪ saved (default: ../../data/interim/HDFS1)
```

FIGURE 4.7: An example of the command-line interface, which enables us to easily use the provided tool.

a logical but flexible project structure for data science tasks. The current version of our project is accessible on GitHub[10].

4.5.2 Data preprocessing and splitting

The Python file `prepare_hdfs.py` provides the interaction with a user via a command-line interface. An example is shown in Figure 4.7. The core functions are implemented in `hdfs.py` and `logparser.py`. Firstly, the HDFS file with labels is loaded into memory. Then, log lines are grouped into blocks by the block ID. The data set is split into training and testing data sets. It is crucial not to divide any block into two sets. Otherwise, it could cause severe information leakage.

We use a stratified splitting technique as the data set is highly imbalanced. The stratified technique guarantees that the percentage of normal and anomalous log blocks is similar in both data sets. This technique is performed at the label level, and thus the number of log lines might differ slightly in individual data sets. However, we are primarily interested in blocks. Therefore, the different number of log lines in data sets does not mean any serious problem. All newly created data sets contain 2.93% anomalous data points thanks to the stratified splitting technique. We also specify a random seed in order to have the splits consistently reproducible.

The `logparser.py` file contains the implementation of Drain3, which is the preprocessing technique used in the baseline methods.

The file `prepare_hdfs.py` also supports k-fold cross-validation. In this case, the file produces multiple pairs of training and validation data sets. Nevertheless, the cross-validation technique is not used in experiments due to the high computational complexity.

We save several output files in this stage, namely the training data set, the validation data set, and the testing data set. These files still contain raw log lines.

[10] `https://github.com/LogAnalysisTeam/methods4logfiles`

4.5.3 From textual to numerical representation

Before we may convert the log lines into numerical vectors, we need a trained fastText model. The fastText library provides a command-line interface, and it can also be used in Python. We leverage Skip-gram model, implemented within the fastText library, and learn the embeddings in an unsupervised manner. The hyperparameters are adopted from [42], and the rest is left with their default values. Specifically, the embedding dimension is 100, and character n-grams are from $n = 3$ to $n = 6$. We could not use multithreading, as the embeddings would not be reproducible.

The Python file `build_features_hdfs.py` ensures proper conversion from textual data to numerical vectors once a trained fastText model is available. Akin to data preprocessing 4.5.2, this file also provides a user-friendly command-line interface. The particular implementation is located in `hdfs.py`. The implementation supports several options. We can access each log line as a standalone data point or a part of a particular block.

The first case involves creating new data labels as the original file with labels contains labels in a per-block manner; recall Figure 4.3. However, this approach is not further assumed, as the HDFS data set is more or less an example of multiple-instance learning (MIL). A model receives a set of instances and only one label. The set is annotated positive if there is at least one anomalous log line. Otherwise, the set is annotated negative. Therefore, it is not sufficient to annotate all log lines corresponding to an anomalous label as anomalies.

The second case creates a list of matrices that contain a numerical representation of a particular block. Additionally, we can append a new feature that tracks time deltas between two succeeding log lines within a block. When a system is inactive, it does not produce any log line. Thus, the logarithm function is applied to time deltas in order to reduce the importance of distant log lines, as suggested in [42]. The timestamp granularity is one second, i.e., a later timestamp may differ by one or more seconds. Therefore, the difference might be equal to zero. We add internally one to all time deltas to prevent the algorithm from undefined behavior. The function responsible for computing time deltas is depicted in Figure 4.8.

The `feature_extractor.py` file processes the data sets prepared for the baseline methods. It contains a class that follows a uniform scikit-learn application programming interface (API). The class implements a transformer, and thus the implementation of `transform` and `fit_transform` methods is required.

The number of occurrences of each log key within a block is calculated and used as a feature vector per a given block. Next, the TF-IDF technique is applied to the data set. Finally, the mean vector is subtracted from all vectors.

When applying this transformation on previously unseen data, two scenarios might happen. Firstly, a new log key was detected by Drain3. In this case, the algorithm skips such keys. Secondly, the unseen data contains only a subset of the log keys encountered in the training data. In this case, corresponding columns are filled with zeros.

The data sets created by the `build_features_hdfs.py` file are saved on a disk in binary format afterward. In contrast, the class, located in the `feature_extractor.py` file, is called on the fly before training a model as it is not a time-consuming transformation.

```
def to_seconds(timedeltas: np.array) -> np.array:
    return np.vectorize(lambda x: int(x.total_seconds()))(timedeltas)

def get_timedeltas(timestamps: np.array) -> np.array:
    # initialize the vector with all ones since log10(0) is undefined
    timedeltas = np.ones(shape=timestamps.shape, dtype=np.int32)
    # we do not lose the information if the time delta is 1
    timedeltas[1:] += to_seconds(timestamps[1:] - timestamps[:-1])
    # decrease the importance of large time differences
    timedeltas = np.log10(timedeltas)
    return timedeltas
```

FIGURE 4.8: Each pair of timestamps is subtracted and converted to seconds. The implementation assures that we do not lose the information whether a particular time delta is equal to one. The common logarithm is applied to the result to decrease the importance of large time differences.

4.5.4 Models

In this subsection, we first describe implemented preprocessing techniques that improve the learning process. Next, we delineate a model prototype that follows the intuitive scikit-learn API.

Feature scaling and cropping

Since we experiment with neural networks, it is crucial to scale the features. It usually speeds up the training [66].

We implemented a modified version of standardization. Many machine learning models often require the standardization of a data set. It centers the data by subtracting the mean vector from each feature vector, and then it scales the data by dividing it by its standard deviation. Standardization is essential for models which make assumptions about the input data. A typical implementation of standardization is shown in Equation 4.5, where μ is the mean and σ is the standard deviation.

The HDFS data set is a list of matrices. Each matrix has a shape given by the number of log lines and the embedding dimension; see Figure 4.9. We first reshape the data set into a 2D matrix, and then the standardization is applied to it.

$$\overline{X} = \frac{X - \mu}{\sigma} \tag{4.5}$$

Akin to the implementation of standardization, the implementation of normalization is also adjusted to the HDFS data set requirements. Normalization scales the data in the range between 0 and 1. It is useful for models which do not assume any distribution of a particular data set, e.g., neural networks. However, normalization is not recommended if the data set contains outliers. Equation 4.6 depicts a common implementation of normalization.

$$\overline{X} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4.6}$$

Our implementations, i.e., standardization and normalization, inherit from the scikit-learn class `StandardScaler` and `MinMaxScaler`, respectively. Furthermore, we

have to implement `fit`, `fit_transform`, and `transform` methods. Besides initial experiments, we decided to use normalization over standardization due to better performance.

The HDFS data set contains blocks with various numbers of log lines; recall Figure 4.2. That is a burden to mini-batch learning. Even though a model supports a variable input, PyTorch does not support a batch of blocks with various numbers of log lines. This limitation leads us to two solutions.

The obvious solution is to use a batch size of size one. Nevertheless, this is not used in practice since the small batch size is less accurate in gradient estimation.

The alternative approach involves padding the data set. As mentioned earlier, the block size is a hyperparameter of a model. Once the block size is known, we crop all blocks with more log lines than the block size. Moreover, we pad the smaller blocks with zeros. An example is shown in Figure 4.9. We used this approach in our experiments, although it might remove some vital information from a particular block.

The implementation of techniques mentioned above, i.e., standardization and normalization, may be found in the `datasets.py` file.

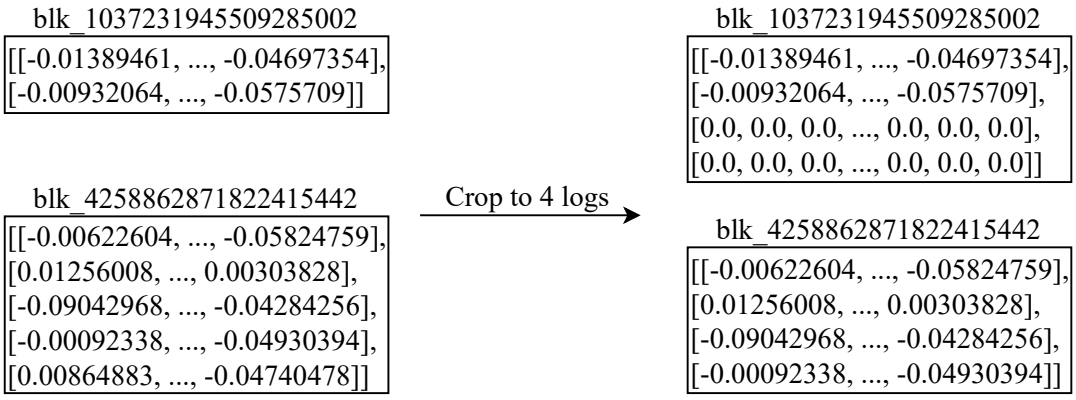


FIGURE 4.9: This figure shows how blocks can be cropped to the desired shape. Let us assume the block size equals 4. The blocks with less than 4 log lines are padded with zeros, and all blocks with more than 4 log lines are cropped to 4 log lines.

Model prototype

We developed and followed a set of guidelines which the implementation of models obey. The scikit-learn library provides some guidance. Compliance with the guidance should enable other researchers to fast prototype training, evaluation, and deployment of a particular model. Each implementation of a model is in a standalone file. One class provides a wrapper around another class that implements a neural network using the PyTorch library.

The wrapper follows best practices as mentioned in scikit-learn. A class should be instantiable without passing any argument into the constructor. Furthermore, each class inherits from the abstract class `OutlierMixin`. It means that methods such as `fit`, `predict`, and `set_params` are implemented. A data set should be a NumPy object which is internally converted to PyTorch tensors.

Once a model is trained on the training data set, the model can predict validation data examples. The predictions are returned in the form of a single float number per data point. Each float number corresponds to a particular reconstruction error of

```
model = AETCN()
model.set_params(**config['hyperparameters'])

model.fit(x_train[y_train == 0]) # train on normal data examples
y_pred = model.predict(x_val) # return the MSE per example

theta, f1_score = find_optimal_threshold(y_val, y_pred)
y_pred = classify(y_pred, theta)
metrics_report(y_val, y_pred)
confusion_matrix(y_val, y_pred)
```

FIGURE 4.10: This example depicts the shared API among all implemented models. Let us assume that data sets were defined beforehand. The AETCN model is instantiated, and the current set of hyperparameters is passed to the model. The model fits the normal training data, i.e., data points annotated as anomalies were removed. Then, the model predicts the MSE of a corresponding validation data point. The decision threshold is estimated, and the data points are classified accordingly.

a data point. We need to find such a threshold that classifies each data point as either normal or anomalous. We optimize the threshold in $\mathcal{O}(n)$ by trying to set each anomalous data point from the validation data set as the threshold. The anomalous data point is called the threshold if and only if it maximizes the F1-score. This threshold is also used once we want to perform the final evaluation of a particular model on the testing data set. A minimalist example of conducting an experiment is depicted in Figure 4.10.

We specify a random seed for all experiments to ensure that our results are reproducible. Nonetheless, the PyTorch library states that completely reproducible results across individual executions are not guaranteed. The primary source of nondeterminism is a graphics card.

4.5.5 RCI Cluster

The RCI Cluster was used to train the models and tune their hyperparameters. It is a gratuitous project for all RCI researchers, and even non-RCI researchers may ask for access to the cluster. The RCI Cluster consists of compute nodes, data storage, and a high-speed network with low latency. The high-speed network interconnects individual compute nodes.

We leveraged graphics cards as most of our models comprise convolutional layers. Some nodes contain multiple instances of NVIDIA Tesla V100 with 32GB graphic memory. This card substantially accelerates the training of deep learning models.

The RCI Cluster uses the SLURM workload manager as a job scheduler. A set of basic commands needed for executing and managing experiments is depicted in Figure 4.11.

```
#!/bin/bash
#SBATCH --partition=gpu
#SBATCH --time=24:00:00
#SBATCH --nodes=1 --ntasks-per-node=1 --cpus-per-task=4
#SBATCH --mem=50G
#SBATCH --gres=gpu:1

# clear the environment from any previously loaded modules
ml purge > /dev/null 2>&1

# train a model
python train_autoencoder.py
```

FIGURE 4.11: An example of a script which reserves a single graphics card and 50 GB of memory. The execution may last at most one day. The script can be added to the `gpu` queue using the command `sbatch task.sh`.

Chapter 5

Evaluation

This chapter delineates our workflow of evaluating the proposed models, and it is divided into several sections. We start with examining the baseline methods, which set the initial thresholds on the metrics. We presented the tuned hyperparameters of the proposed models in Section 4.4. These models are evaluated on validation and testing data sets. We include a detailed analysis of the performance using various metrics. Finally, we provide some insights into the obtained results.

The complexity of models prolongs the time needed for model training. Although the search spaces are enormous, we had to compromise the number of trials in a particular experiment. We tried to find a trade-off between the number of trials and the execution time of an experiment. Therefore, each experiment using a random search performs 100 trials on a particular search space. This number of trials does not hold for a grid search where all predefined combinations are evaluated.

5.1 Baseline models

Baseline models expect a one-dimensional vector with 48 real numbers as the input. The numerical vector aggregates the information from all log lines belonging to a particular block. The obtained vectors are centered by subtracting the mean vector and normalized by the weighting method TF-IDF. The approach is described in more detail in Subsection 4.2.1.

The Local Outlier Factor and Isolation Trees models have a small number of hyperparameters. Therefore, we tuned the hyperparameters on a grid search using predefined values. On the other hand, the vanilla autoencoder leverages a random search as the number of hyperparameters significantly increases. All baseline models are delineated in Subsection 4.4.2.

5.1.1 Local Outlier Factor model

The LOF model has two crucial hyperparameters, namely the number of neighbors and the distance metric. The most promising configuration sets the number of neighbors to 450 and uses the Chebyshev metric. The performance on the validation and testing data set is depicted in Table 5.1. This model performed poorly compared to other baseline methods since it has scalability issues in high-dimensional space as mentioned by the authors [67]. Therefore, this model is not assumed as a candidate in hybrid models due to a rise in the input dimension.

	Precision	Recall	F1-score
Validation data set	0.4350	1.0000	0.6062
Testing data set	0.4288	0.9281	0.5866

TABLE 5.1: Detailed metrics of the LOF model with tuned hyperparameters.

5.1.2 Isolation Trees model

The more sophisticated baseline model is Isolation Trees. This model comprises three essential hyperparameters, namely the number of trees, the number of examples drawn from a data set to train a tree, and the number of features drawn from all available features to train a tree. The optimal model has 115 trees, uses 32 features but only 1% of the training examples. The total number of available features is 48, i.e., the model utilizes 66.7% of the provided features. The training data set consists of 465,798 blocks. Interestingly, the model leverages only 4,657 blocks to train individual trees. The most optimal percentage of training examples might slightly differ since we used a grid search for hyperparameter tuning. Table 5.2 shows the achieved performance on both validation and testing data sets. The Isolation Trees model significantly outperformed the LOF model and achieved similar performance to other embedding-based models.

	Precision	Recall	F1-score
Validation data set	0.8157	0.8152	0.8155
Testing data set	0.8095	0.8076	0.8086

TABLE 5.2: Detailed metrics of the Isolation Trees model with tuned hyperparameters.

5.1.3 Vanilla autoencoder

Once we explored the performance of baseline unsupervised machine learning models on the HDFS data set, we wanted to prove our idea that semi-supervised machine learning models can outperform the baseline models. We implemented a vanilla autoencoder that uses the same preprocessing of the HDFS data set as the other baseline models. We tuned hyperparameters of the autoencoder using a random search due to the immense number of hyperparameters.

The optimal architecture comprises six fully connected layers. The encoder has only one hidden layer with 183 neurons. The bottleneck compresses the information into 66 neurons. Finally, the decoder decodes the compressed information back to the vector with 48 numerical values. The decoder consists of hidden layers with 115, 151, 171, 192, and 48 neurons. The model was trained for two epochs with the learning rate 0.0013. The batch size was set 16, and the dropout was 0.0729, i.e., each neuron is skipped with approximately 7% probability. The decision threshold is 0.1530. The smaller the decision threshold is, the better the model can reconstruct an arbitrary input.

The performance of the vanilla autoencoder is depicted in Table 5.3. This model performed significantly better than the other baseline models. Moreover, it achieved

the highest F1-score on both the validation and the testing data sets from all proposed models. Surprisingly, the vanilla autoencoder has the lowest number of trainable parameters among the proposed neural networks, as shown in Table 5.15.

	Precision	Recall	F1-score
Validation data set	0.9301	0.8165	0.8696
Testing data set	0.9314	0.8302	0.8779

TABLE 5.3: Detailed metrics of the vanilla autoencoder model with tuned hyperparameters.

Vanilla autoencoder with aggregated embeddings

We also experimented with aggregated blocks of embeddings produced by the trained fastText model. The idea comes originally from multiple-instance learning. In this setup, we tried to reconstruct an aggregated vector. The reconstruction of the original block would be much more difficult as an aggregation function removes some information and the order of log lines. We proposed two different aggregation functions, namely maximum and average. Each block of embeddings is first aggregated to a one-dimensional vector accordingly. Then the vanilla autoencoder tries to learn to copy the numerical vector to its output.

Although we ran an extensive random search on the hyperparameters, we were not able to achieve any promising results. The best models with different aggregation functions achieved a similar F1-score on the validation data set — with the maximum aggregation function 0.5525, and with the average aggregation function 0.5518, precisely. The models could not distinguish between the normal and anomalous classes as both aggregation functions probably removed a substantial amount of information needed for the accurate distinction. We did not explore this direction further due to the disappointing results. Moreover, these experiments were conducted to help us to better understand the baseline performance on the data set.

5.2 Embedding-based models

Once we had well-established baseline performance on the metrics, we started thinking about models capable of handling a matrix as the input and ideally producing another matrix with the same shape as the output. Naturally, we decided to research convolution-based methods at first. The ideal candidate is a model with TCN blocks as it leverages convolutional layers and can produce a sequence in the output.

5.2.1 TCN model

A TCN model usually outperforms classical LSTM approaches for sequence-to-sequence tasks. The TCN model can process a sequence of an arbitrary length and outputs a new sequence but with the same length as the input. The TCN block contains two 1D convolutional layers followed by the ReLU activation function and the dropout.

The first experiment continues in the research proposed by authors [14] available on GitHub[1]. Initially, the number of feature maps in each TCN block was random. Unfortunately, this model performed poorly in terms of the F1-score, precisely 0.5514. A key characteristic is that the output at index i can leverage entries of an input sequence that occur before i. Nevertheless, we disposed of this limitation in order to let each neuron utilize the whole block, not just the preceding log lines. This approach is known as bidirectional TCN in literature [68]. The models with this modified architecture achieved negligible improvement in all metrics. There arose an idea that the model could not properly compress the information. Therefore, we proposed an architecture that follows the pattern of autoencoders, i.e., the first few blocks compress the input, then the succeeding blocks try to reconstruct the input back. We controlled the compression and the decompression using the number of feature maps in the individual TCN blocks. Nevertheless, the autoencoder-based architecture did not improve the performance of the model on the metrics.

The autoencoder-based model achieved the best performance with the following properties — the batch size is set to 32, the dropout is 0.1942, the kernel size of convolutional blocks is 9. The architecture contains four TCN blocks with 100, 895, 250, and 100 feature maps. We tuned the input block size during the hyperparameter optimization. The optimal block size utilizes the first 12 log lines. The model was trained for seven epochs with the learning rate 0.0001. The decision threshold is 0.0000098, which means that the model could almost learn the identity function completely. The performance of the model is depicted in Table 5.4. The TCN model performed worse than any baseline method.

	Precision	Recall	F1-score
Validation data set	0.9966	0.3842	0.5545
Testing data set	0.9935	0.3646	0.5334

TABLE 5.4: Detailed metrics of the autoencoder-based TCN model with tuned hyperparameters.

5.2.2 AETCN model

The TCN model did not meet our expectations. Therefore, we proposed a modified version of the TCN model where the model explicitly contains a bottleneck. This bottleneck is implemented as a fully connected layer which helps the model to further compress the information. The AETCN model comprises an encoder (TCN blocks), the bottleneck (a fully connected layer), and a decoder (TCN blocks). We observed a significant improvement on the metrics with this architecture.

The optimized architecture consists of one TCN block with 142 feature maps (the encoder), the bottleneck with 1246 neurons, and one TCN block with 100 feature maps (the decoder). The convolutional layers use the kernels of size 3. The dropout is set to 0.3239, and the batch size equals 8. The block size uses 45 log lines which the majority of the blocks fulfill; see Figure 4.2. We trained the model for four epochs with the learning rate 0.0016. The decision threshold is equal to 0.0033. The results on the metrics are depicted in Table 5.5. The AETCN model achieved auspicious performance on the F1-score metric. However, the model suffered from overfitting

[1] `https://github.com/locuslab/TCN`

as the decrease in the F1-score on the testing data set is the highest among all proposed models. Moreover, the TCN-based models have a typically greater number of trainable parameters by order of magnitude than other models; see Table 5.15.

Finally, we examined the bottleneck in more detail. On average, only 35 neurons are active after applying the ReLU activation function to the fully connected layer, i.e., approximately 97% of neurons are nullified.

	Precision	Recall	F1-score
Validation data set	0.8794	0.8040	0.8400
Testing data set	0.8576	0.7797	0.8168

TABLE 5.5: Detailed metrics of the AETCN model with tuned hyperparameters.

5.2.3 CNN1D model

A more suitable layer, in terms of flexibility, is a convolutional layer. Each convolutional layer reduces the dimension of input. Moreover, there exists a complement to each convolutional layer called the transposed convolutional layer. The transposed convolutional layer enlarges an input. Therefore, we may create an architecture that allows us to use a matrix as the input and still follows the autoencoder pattern. We leveraged 1D convolutional and transposed convolutional layers in the architecture of this model.

Our convolutional block comprises one 1D convolutional layer, followed by the activation function ReLU and the 1D max-pooling layer. On the other hand, the transposed convolutional block contains one 1D transposed convolutional layer, followed by the activation function ReLU and an upsampling layer. The transposed convolutional block is the complement of the convolutional block.

The most optimized model consists of one convolutional block with 195 feature maps (the encoder) and three transposed convolutional blocks with 170, 164, and 100 feature maps (the decoder). Since we used convolution-based layers in both the encoder and the decoder, we tuned the kernel size of the encoder and the kernel size of the decoder separately. The optimal value of the kernel size in the convolutional layer is 3, and the kernel size in the transposed convolutional layers is 11.

Each convolutional block reduces the dimension of a particular input, and therefore the block size is dependent on the number of convolutional layers and the kernel size precisely. The number of the block size is determined randomly in the range from `min_block_size` to `min_block_size + 32`. The `min_block_size` variable represents the smallest number of log lines that produces feature maps with positive dimensions, and 32 is an arbitrarily chosen number that increases the variety of possible choices. The block size equals 32 log lines in the model. The model was learned for three epochs with the learning rate 0.0006 and the batch size 128. The decision threshold is 0.0036. Table 5.6 depicts the performance of the CNN1D model on the metrics. This model achieved a similar performance on both the validation data set and the testing data set as the Isolation Trees model.

5.2.4 AECNN1D model

Akin to the AETCN model, we proposed an architecture that enhances the CNN1D model by an additional fully connected layer. The architecture comprises several

	Precision	Recall	F1-score
Validation data set	0.9511	0.7320	0.8273
Testing data set	0.9292	0.7090	0.8043

TABLE 5.6: Detailed metrics of the CNN1D model with tuned hyperparameters.

1D convolutional blocks, the bottleneck, and several 1D transposed convolutional blocks. The bottleneck is the only difference from the CNN1D model.

The architecture with tuned hyperparameters consists of two 1D convolutional blocks with 220 and 409 feature maps (the encoder). The kernel size of the convolutional layers is 7. Next, the bottleneck has 945 neurons. Finally, three 1D transposed convolutional blocks form the decoder. The number of feature maps in individual blocks is 405, 58, and 100, respectively. The kernel size of the transposed convolutional layers is 5. The model expects 28 log lines in the input. We trained the model for eight epochs with the learning rate 0.0007. The batch size was experimentally set to 8. The decision threshold is 0.0075, and the performance of this model on the metrics is shown in Table 5.7. Moreover, the histogram of the reconstruction error on the testing data set is depicted in Figure 5.1.

The AECNN1D model confirmed that adding a fully connected bottleneck helps the model to compress the information thoroughly. In addition, this model achieved the highest F1-score metric on the testing data set among all embedding-based models.

	Precision	Recall	F1-score
Validation data set	0.8274	0.8799	0.8528
Testing data set	0.8447	0.8753	0.8597

TABLE 5.7: Detailed metrics of the AECNN1D model with tuned hyperparameters. Interestingly, the model performed slightly better on the testing data set than on the validation data set in terms of the F1-score.

5.2.5 CNN1DTCN model

As both the CNN1D model and the AETCN model showed promising results, we proposed their combination. The idea behind the CNN1DTCN model is as follows. Initially, 1D convolutional blocks extract local dependencies. Next, TCN blocks are plugged in as they can forward the information to all output neurons. Moreover, TCN might be viewed as the bottleneck of the model. Finally, 1D transposed convolutional blocks reconstruct the information back to the same shape as the input.

The best architecture contains three 1D convolutional blocks, three TCN blocks, and three 1D transposed convolutional blocks. The individual blocks consist of 68, 115, and 486 feature maps (1D convolutional blocks), 98, 394, and 498 feature maps (TCN blocks), 497, 275, and 100 feature maps (1D transposed convolutional blocks). The kernel size of convolutional layers is 3, and the kernel size of transposed convolutional layers is 13. The TCN blocks have the dropout with 0.3087 probability of skipping a particular connection between neurons. The batch size is 64. The model

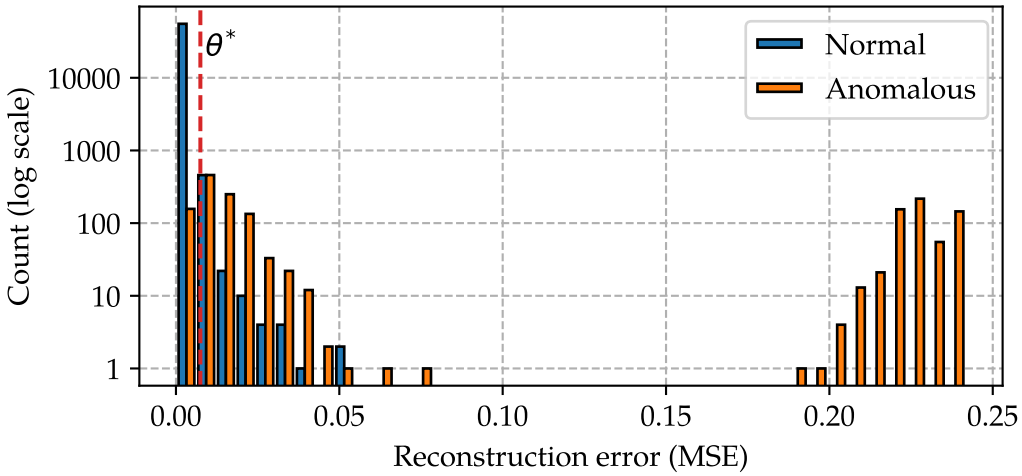


FIGURE 5.1: The histogram of the reconstruction error on the testing data set. The reconstruction error is computed as the MSE (Equation 4.4). The decision threshold θ^* had been estimated on the validation data set, and then the threshold was used for the classification. All data points with greater MSE than θ^* are classified as anomalies.

was trained for six epochs with the learning rate 0.0005. The block size was experimentally set to 49 log lines, which most blocks fulfill and only a minority of blocks have to be cropped. The optimal decision threshold is 0.0029. Table 5.8 shows the performance of the model on the metrics. The achieved F1-scores on validation and testing data sets showed that the combination of TCN blocks and 1D convolutional layers forming an autoencoder-based model could perform better than the individual CNN1D or TCN models.

	Precision	Recall	F1-score
Validation data set	0.9549	0.7406	0.8342
Testing data set	0.9454	0.7197	0.8173

TABLE 5.8: Detailed metrics of the CNN1DTCN model with tuned hyperparameters.

5.2.6 CNN2D model

Naturally, once we explored the possibilities of 1D convolutional layers, we focused on 2D convolutional layers. The main difference is in the shape of an input matrix. A particular kernel of the 1D convolutional layer convolves across the log lines, and the embedding dimension is viewed as the number of input channels. In contrast, the 2D convolutional layer convolves across the whole matrix, i.e., both the log lines and the embedding dimension. Therefore, the kernel size is not a single integer but a tuple of two numbers. The dimension of input channels must be artificially added, and it equals one. Therefore, the input shape of a standalone tensor has three dimensions. The 2D convolutional and 2D transposed convolutional blocks look similar to 1D convolutional and 1D transposed convolutional blocks. The only difference is that 1D convolutional and 1D transposed convolutional layers are substituted with 2D convolutional and 2D transposed convolutional layers, respectively.

The model with tuned hyperparameters has six blocks, three 2D convolutional blocks, and three 2D transposed convolutional blocks. The 2D convolutional blocks consist of 28, 199, and 250 feature maps (the encoder). On the contrary, the 2D transposed convolutional blocks consist of 236, 131, and 1 feature map (the decoder). Akin to the last blocks in other models, the last block has to be in agreement with the input number of feature maps. The kernel size of the 2D convolutional blocks is a tuple with the height 5 and the width 3. The kernel size of the 2D transposed convolutional blocks is a tuple with dimensions 15 and 7. The batch size is 32, and the block size is 41 log lines. We trained the model for four epochs with the learning rate 0.0005. The decision threshold is 0.0046.

The performance of the model is depicted in Table 5.9. Surprisingly, adding a fully connected layer as the bottleneck did not improve the performance of the model on the metrics. Therefore, we did not further explore such architecture. Moreover, the models with 2D convolutional layers performed generally worse than the models leveraging 1D convolutional layers. The CNN1D model also has a lower number of trainable parameters by order of magnitude than the CNN2D model, as depicted in Table 5.15.

	Precision	Recall	F1-score
Validation data set	0.9172	0.7234	0.8089
Testing data set	0.8943	0.7084	0.7906

TABLE 5.9: Detailed metrics of the CNN2D model with tuned hyperparameters.

5.2.7 SACNN1D

Once we exhausted all possibilities of convolutional layers, we concentrated on transformer-based models. The main building block of a transformer is the self-attention mechanism; recall Section 2.3. This layer is designed directly to work with sequences. Therefore, no additional changes to the architecture were required.

The optimized architecture comprises two 1D convolutional blocks with 128 and 269 feature maps, respectively. The kernel size of each convolutional block is 7. Next, the 269-head self-attention layer follows. Finally, three 1D transposed convolutional blocks with the 1-head self-attention layer after the second 1D transposed convolutional block follow. The kernel size of each 1D transposed convolutional block is 9. The self-attention layer uses the dropout, and the dropout is set to 0.2891 for both self-attention layers. The batch size is set to 64. The model utilizes 35 log lines for the input. The model was trained for six epochs with the learning rate 0.0007. The decision threshold is set to 0.0026. The performance of the model is shown in Table 5.10. The model achieved almost the identical performance as the CNN1D model on the validation data set, but the SACNN1D model generalized better than the CNN1D model on the testing data set. Furthermore, the difference in the numbers of trainable parameters of both comparing models is negligible; see Table 5.15.

5.2.8 SACNN2D

Exploring the 2D self-attention layer comes quite naturally once we experimented with the 1D self-attention mechanism. Unfortunately, 2D self-attention mechanism is not straightforward. Especially, handling matrices instead of sequences makes it

	Precision	Recall	F1-score
Validation data set	0.9442	0.7373	0.8280
Testing data set	0.9300	0.7179	0.8103

TABLE 5.10: Detailed metrics of the SACNN1D model with self-attention layers and tuned hyperparameters.

difficult. We were inspired by the authors [18, 64]. Our proposed architecture of the 2D self-attention layer is depicted in Figure 4.5. Since the architecture is complex, we used the 2D self-attention layer as the bottleneck of the model. The bottleneck is controlled by the number of feature maps in the 2D self-attention layer. The 2D convolutional and 2D transposed convolutional blocks are the same as in the CNN2D model.

The optimal architecture of the SACNN2D model consists of two 2D convolutional blocks with 204 and 328 feature maps, respectively. Next, the 2D self-attention layer with 81 feature maps follows. Furthermore, the model contains two 2D transposed convolutional blocks with 80 and 1 feature maps, respectively. The kernel size of 2D convolutional blocks is a two-dimensional tuple — 5 and 3, precisely. The kernel size of 2D transposed convolutional blocks is also a two-dimensional tuple — 7 and 9. The block size is 29 log lines, and the batch size is 8. We trained the model for four epochs with the learning rate 0.0012. The decision threshold is 0.0150, which suggests itself that the model cannot efficiently reconstruct the input. The performance of the model on the metrics also supports this claim; see Table 5.11. The SACNN2D model achieved the worst performance on both validation and testing data sets among all proposed embedding-based models; see Table 5.14.

	Precision	Recall	F1-score
Validation data set	0.9682	0.6640	0.7878
Testing data set	0.9470	0.6479	0.7694

TABLE 5.11: Detailed metrics of the SACNN2D model with the 2D self-attention layer and tuned hyperparameters.

5.3 Hybrid models

We used the AETCN model for feature extraction. The input of hybrid models is the output of the bottleneck from a particular trained instance of the AETCN model. The bottleneck may be viewed as newly compressed embeddings. We did not use the AETCN model, which achieved the highest performance on the metrics, but rather an instance of the AETCN model with a relatively low-dimensional bottleneck. Nevertheless, the chosen model did not achieve statistically significantly worse performance than the best-reported model; see more details of the tuned AETCN model in Subsection 5.2.2.

The output bottleneck dimension of the used the AETCN model equals 106. We leveraged the features obtained before applying the ReLU activation function. Otherwise, only 35 neurons would remain non-zero on average, i.e., approximately 67% of neurons are nullified. The obtained features are always scaled such that the mean

equals zero and the standard deviation is equal to one. Next, a particular model is trained on the scaled features.

Finally, we started with the Isolation Trees model and compared its performance with a vanilla autoencoder.

5.3.1 Isolation Trees and AETCN

We used the features from the AETCN bottleneck as the input to the Isolation Trees model. Since the dimension of the newly created embeddings is greater than the dimension of the baseline (bag-of-words) approach, we hoped that the new features might capture more information.

The optimized unsupervised model uses 97 out of 106 features, has 645 trees, and is fit only on 3% of the training examples. The training data set consists of 465,798 blocks. Interestingly, the model leverages only 13,973 blocks to train individual trees. Nevertheless, the most optimal hyperparameters might be slightly different since we used a grid search for hyperparameter tuning. Table 5.12 shows the achieved performance on the validation and testing data sets. The performance is worse than the performance achieved by the standalone Isolation Trees model. The issues might arise from the loss of relevant information in the bottleneck of the AETCN model or high-dimensional input data.

	Precision	Recall	F1-score
Validation data set	0.8168	0.5122	0.6296
Testing data set	0.7964	0.5018	0.6157

TABLE 5.12: Detailed metrics of the tuned hybrid model, which comprises the AETCN model and the Isolation Trees model.

5.3.2 Vanilla autoencoder and AETCN

Akin to the previous hybrid model, information gain was expected as the newly created embeddings have a higher dimension. Moreover, the idea of building an autoencoder on top of one or multiple autoencoders has been thoroughly studied and showed promising results [65].

The hyperparameters of this hybrid model were tuned using a random search since the number of hyperparameters is enormous. The optimal model consists of an encoder and a decoder. The encoder contains fully connected layers with 453, 452, 447, 346, 201, and 116 neurons. The decoder comprises three fully connected layers with 125, 478, and 106 neurons. The batch size equals 32, and the dropout is set to 0.0218. We trained the model for eight epochs with the learning rate 0.0002. The decision threshold is equal to 0.2052. The performance of the model is depicted in Table 5.13. Akin to the previous hybrid model, the performance is worse than the performance achieved by the standalone vanilla autoencoder.

5.4 Discussion

The initial experiments showed that unsupervised models could compete with semi-supervised models, namely autoencoder-based models, in terms of the F1-score. The Isolation Trees algorithm significantly outperforms the LOF model on both validation and testing data sets. Moreover, the Isolation Trees model performs similarly

	Precision	Recall	F1-score
Validation data set	0.9125	0.6950	0.7891
Testing data set	0.9013	0.6829	0.7770

TABLE 5.13: Detailed metrics of the tuned hybrid model, which comprises the AETCN model and the vanilla autoencoder model.

to multiple models which utilize embeddings created by the trained fastText model, see Table 5.14. The vanilla autoencoder, Isolation Trees, and LOF models leverage bag-of-words representation as the input of a particular model. The vanilla autoencoder achieved the best F1-score on the testing data set, 0.8779 precisely. It emerged that this was the highest reported F1-score overall.

Model	Validation F1-score	Testing F1-score
Local Outlier Factor	0.6062	0.5866
Isolation Trees	0.8155	0.8086
Vanilla autoencoder	**0.8696**	**0.8779**
TCN	0.5545	0.5334
CNN1D	0.8273	0.8043
CNN2D	0.8089	0.7906
CNN1DTCN	0.8342	0.8173
AETCN	0.8400	0.8168
AECNN1D	**0.8528**	**0.8597**
SACNN1D	0.8280	0.8103
SACNN2D	0.7878	0.7694
Isolation Trees and AETCN	0.6296	0.6157
Vanilla autoencoder and AETCN	0.7891	0.7770

TABLE 5.14: This table shows the results on the validation and testing data sets per each model. The orange color highlights the baseline models. On the other hand, the blue color highlights the hybrid models. The green color highlights the model with the overall best performance on the HDFS data set. At the same time, the purple color highlights the most promising model which operates with embeddings.

Next, the TCN model can almost learn the identity function, but it is not desirable as we need a model capable of distinguishing normal and anomalous classes. The performance of the TCN model is 0.5334, which is the worst reported result in this thesis. A characteristic feature of the TCN-based models is the complexity and the number of trainable parameters; see Table 5.15. Three out of four models with the highest number of trainable parameters are the TCN-based models. However, the TCN-based models are beneficial with the fully connected bottleneck. The AETCN model outperforms most of the proposed models on the validation data set with the F1-score equal to 0.8400. Nevertheless, the difference between the F1-score on the validation data set and the testing data set is the highest among all models. Specifically, the decrease in the F1-score is 0.0232.

The CNN1D model outperforms the CNN2D model. Moreover, the CNN1D has approximately ten times fewer trainable parameters than the CNN2D model;

see Table 5.15. The answer to the diverse performance might be found fundamentally in convolutional layers. The 1D convolutional kernel convolves only across log lines. The embedding dimension is viewed as a number of input channels. Even though the input is a two-dimensional matrix, only the dimension, which represents log lines, is spatial. On the contrary, the 2D convolutional kernel convolves across both dimensions. A fully connected bottleneck further boosts the performance of the CNN1D model, which we named the AECNN1D model. The AECNN1D model achieved the highest F1-score of all models operating with embeddings.

We also explored the models with the self-attention mechanisms, as many state-of-the-art architectures in the NLP domain incorporate self-attention layers. The traditional self-attention layer proposed by authors [16] can fully substitute the 1D convolutional layer once certain conditions are met [20, 22]. Adapting the self-attention layer still shows room for further improvement, such as a complete replacement of 1D convolutional layers with self-attention layers. The SACNN1D model achieved a statistically similar F1-score to other models leveraging the 1D convolutional layer; see Table 5.14. On the contrary, the SACNN2D model could not compete with other models, which has several explanations. Firstly, the 2D convolutional layer might not be suitable for log files, as mentioned above. Secondly, the proposed architecture contains only one self-attention layer, which is very complex. Therefore, the self-attention mechanism for higher dimensional data needs further investigation.

Finally, we proposed models which leverage the AETCN model. The bottleneck extracts essential features which are further used as a one-dimensional input vector for another model. We tried first the Isolation Trees model since it achieved high performance on the testing data set. Akin to the Isolation Trees model, we also trained the vanilla autoencoder. Neither the unsupervised model (Isolation Trees) nor the semi-supervised model (vanilla autoencoder) achieved the F1-score comparable to other models; see Table 5.14. The explanation might involve the sparse feature vector — recall that, on average, only 35 neurons remain non-zero after applying the ReLU activation function. We may use the AECNN1D for feature extraction instead of the AETCNN model in the future.

Model	Number of parameters
TCN	11,250,450
CNN1D	910,859
CNN2D	10,286,030
CNN1DTCN	7,938,214
AETCN	16,136,818
AECNN1D	3,307,510
SACNN1D	1,287,178
SACNN2D	2,772,356
Vanilla autoencoder	114,612
Vanilla autoencoder and AETCN	829,984

TABLE 5.15: This table shows the number of trainable parameters per each proposed neural network. The orange color highlights a baseline model. On the other hand, the blue color highlights a hybrid model.

Chapter 6

Conclusion

Anomaly detection methods can monitor the behavior of a distributed system in a data center and can detect an anomalous state of a particular machine. With the current demand for cloud services, providing highly available and error-free services has become increasingly important. High availability of services can be achieved by quickly identifying the root cause, and generating logs is one possible solution. Nowadays, no company can afford to inspect log files manually. Therefore, there is a surge of interest in developing methods which can automatically detect anomalous behavior with high accuracy.

Our intensive research confirmed that many existing solutions still depend on log parsing. The main disadvantage of this approach is caused by not using the whole log line but rather only an extracted log key. Moreover, there arises a problem of handling unseen log keys, which might appear in a testing data set. However, other approaches exist which utilize the information hidden in semi-structured data, such as log files. Therefore, we leveraged a trained fastText model, which converts a particular log line into a continuous vector representation, known as a sentence embedding. This approach eliminates the limitation of using only an extracted log key. Furthermore, the fastText library can generalize the underlying semantics. It can create an embedding vector even for an unseen log line, and thus there is no need to retrain the model frequently.

We described the HDFS data set, which contains more than 11 million log lines. We proved the difficulty of the anomaly detection task applied to log files by the first baseline experiments. The baseline methods use a simplified feature extraction based on log parsing and the weighting TF-IDF technique. We selected the Isolation Trees and the Local Outlier Factor models and implemented a vanilla autoencoder as a more advanced baseline method. The hyperparameters of the Isolation Trees and the Local Outlier Factor models were tuned using a grid search. However, due to the immense number of hyperparameters of the vanilla autoencoder, we tuned its hyperparameters using a random search. Next, we proposed novel autoencoder-based models leveraging the information provided by the trained fastText model. We comprehensively evaluated 13 different autoencoder-based models and conducted additional experiments, which did not achieve promising results. The models comprise the temporal convolutional layers, the convolutional and transposed convolutional layers, or the self-attention mechanism. Moreover, we experimented also with hybrid models, which combine the AETCN model with another model. Each model was trained on the training data set, and its hyperparameters were exhaustively optimized using a random search. The hypothesis assumes that the underlying probability distribution of normally-labeled samples retains the specific information distinguishing between the normal and anomalous classes. Therefore, the models were trained in a semi-supervised manner.

The vanilla autoencoder trained on the simplified extracted features achieved the overall best performance on the testing data set — the F1-score was equal to 0.8779. On the contrary, the AECNN1D model outperformed all other proposed models with the F1-score equals 0.8597 on the testing data set. The performance of both mentioned models is comparable regarding the achieved F1-score. Moreover, the AECNN1D model utilizes the embedding representation and is generally applicable to deploying into the production since no additional requirements or periodic retraining is necessary.

Future work may extend the proposed models by further research in the NLP domain. We believe that the self-attention mechanism still poses new room for further improvements. In addition, we initialized experiments with contextual embeddings. Unfortunately, the early results did not show any exceptional embeddings, which would provide more information than fastText embeddings. Finally, the concept of transformer-based models might also be worth considering.

Bibliography

[1] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

[4] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[5] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[7] Andrej Karpathy. CS231n convolutional neural networks for visual recognition. `http://cs231n.github.io`, 2021. Accessed: 7.4.2021.

[8] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320, 2005.

[9] Martin Koryťák. Operating speed estimation on road segments. Bachelor's thesis, Czech Technical University in Prague, 2018.

[10] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.

[11] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[12] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *arXiv preprint arXiv:1706.02515*, 2017.

[13] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[14] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv:1803.01271*, 2018.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[17] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. In *International conference on machine learning*, pages 7354–7363. PMLR, 2019.

[18] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7794–7803, 2018.

[19] Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V Le. Attention augmented convolutional networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3286–3295, 2019.

[20] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jonathon Shlens. Stand-alone self-attention in vision models. *arXiv preprint arXiv:1906.05909*, 2019.

[21] Lilian Weng. Attention? Attention! *lilianweng.github.io/lil-log*, 2018.

[22] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584*, 2019.

[23] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

[24] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *arXiv preprint arXiv:2003.05991*, 2020.

[25] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[26] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[27] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

[28] Prasanta Gogoi, DK Bhattacharyya, Bhogeswar Borah, and Jugal K Kalita. A survey of outlier detection methods in network anomaly identification. *The Computer Journal*, 54(4):570–588, 2011.

[29] Mohiuddin Ahmed, Abdun Naser Mahmood, and Md Rafiqul Islam. A survey of anomaly detection techniques in financial domain. *Future Generation Computer Systems*, 55:278–288, 2016.

[30] Mohammad Braei and Sebastian Wagner. Anomaly detection in univariate time-series: A survey on the state-of-the-art. *arXiv preprint arXiv:2004.00433*, 2020.

[31] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 583–588. IEEE, 2007.

[32] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[33] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 36–43. IEEE, 2004.

[34] Bernhard Schölkopf, Robert C Williamson, Alexander J Smola, John Shawe-Taylor, John C Platt, et al. Support vector method for novelty detection. In *NIPS*, volume 12, pages 582–588. Citeseer, 1999.

[35] Chong Zhou and Randy C Paffenroth. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 665–674, 2017.

[36] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 93–104, 2000.

[37] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 eighth ieee international conference on data mining*, pages 413–422. IEEE, 2008.

[38] Raghavendra Chalapathy and Sanjay Chawla. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407*, 2019.

[39] Amir Farzad and T Aaron Gulliver. Unsupervised log message anomaly detection. *ICT Express*, 6(3):229–237, 2020.

[40] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

[41] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[42] Marek Souček. Log anomaly detection. Master's thesis, Czech Technical University in Prague, 2020.

[43] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[44] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[45] Risto Vaarandi and Mauno Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*, pages 1–7. IEEE, 2015.

[46] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[48] Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[49] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010.

[50] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019.

[51] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI*, volume 7, pages 4739–4745, 2019.

[52] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[53] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817, 2019.

[54] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[55] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111. IEEE, 2016.

[56] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. Detecting anomaly in big data system logs using convolutional neural network. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 151–158. IEEE, 2018.

[57] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[58] Bartosz Krawczyk. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence*, 5(4):221–232, 2016.

[59] Martin Koryťák and Thomas Parnell. Accelerated training using Snap ML. `https://www.kaggle.com/arozrl/accelerated-training-using-snap-ml`, 2020. Accessed: 7.4.2021.

[60] Yuzhe Yang and Zhi Xu. Rethinking the value of labels for improving class-imbalanced learning. *arXiv preprint arXiv:2006.07529*, 2020.

[61] Yanmin Sun, Andrew KC Wong, and Mohamed S Kamel. Classification of imbalanced data: A review. *International journal of pattern recognition and artificial intelligence*, 23(04):687–719, 2009.

[62] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[63] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[64] Meng Li, William Hsu, Xiaodong Xie, Jason Cong, and Wen Gao. Sacnn: self-attention convolutional neural network for low-dose ct denoising with self-supervised perceptual loss network. *IEEE transactions on medical imaging*, 39(7):2289–2301, 2020.

[65] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.

[66] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[67] Timothy De Vries, Sanjay Chawla, and Michael E Houle. Finding local anomalies in very high dimensional space. In *2010 IEEE International Conference on Data Mining*, pages 128–137. IEEE, 2010.

[68] Jian Sun, Wu Guo, Bin Gu, and Yao Liu. Bidirectional temporal convolution with self-attention network for ctc-based acoustic modeling. In *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 1262–1266. IEEE, 2019.