Matthias Krawanja, BSc

Detecting Change in User Behavior with Log Files from Web Servers

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor
Dipl.-Ing. Herbert Leitold
Dipl.-Ing. Dominik Ziegler

Assessor
Univ.-Prof. Dipl-Ing. Dr.techn. Stefan Mangard

Institute of Applied Information Processing and Communications

Faculty of Computer Science and Biomedical Engineering

Graz, May 2020

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Acknowledgements

At this moment, I want to express my wholehearted thankfulness to all who accompanied and supported me throughout the road of my studies.

Foremost, I would like to thank my parents Regina and Arnold, for encouraging me to fight for my goals and always supporting me in my life.
Thank you!

I wish to express my deepest gratitude to my supervisor Dominik. His excellent guidance and priceless feedback throughout every phase of this thesis arouse enthusiasm in me to give my best. Additionally, I want to thank him for reviewing the draft of this thesis.

I also wish to thank all my friends for their companionship during fun times, but also during challenging ones. They were the ones who motivated me to push my limits, while also teaching me to enjoy the precious time spent together.

Furthermore, I owe my thanks to Sandra, my team leader at Siemens, for the opportunity to join her team. They gave me much helpful advice on how to proceed with this thesis. Discussions during coffee breaks were informative and fun. It always felt easier to focus again afterward.

Additionally, I want to thank Claudia, Hartmut, and their team at Arland. They allowed me to take foot in the business world while also starting my academic journey. Their flexibility, whenever required by the university, was priceless.

This thesis is based upon work supported by Google Cloud.

Abstract

Modern systems designed to protect Application Programming Interfaces (APIs) do not monitor changes in user behavior. As a result, they typically do not perform well when dealing with attacks related to credential abuse. Protecting against such attacks requires a fundamentally different approach. Systems must inspect sequences of HTTP requests, instead of focusing on individual ones. Additionally, the amount of data processed by APIs has increased in recent years. As a result, machine learning has gained in popularity.

In this thesis, we present a novel system that detects anomalies in user behavior patterns. Our approach relies on a Deep Neural Network (DNN), set up as an Autoencoder. The proposed system detects anomalies by measuring the reconstruction error of the request sequences. Existing work typically analyses the full HTTP requests. In contrast, our work solely relies on log files, which contain only a fraction of the available information. As a result, our approach does neither depend on information from the client's computer, nor does it need a list of possible HTTP requests beforehand.

We evaluate our system based on a log file from an online store. The results support the claim that the amount of information preserved in log files is sufficient to learn behavioral patterns. We also verify that the approach of first compressing the information and then modeling user behavior works with log entries. Finally, we show that finding a well-performing trade-off between *detection of all attacks (recall)* and *not raising too many false alarms (precision)* is crucial.

Kurzfassung

Moderne Systeme zum Schutz von Application Programming Interfaces (APIs) überwachen Veränderungen im Benutzerverhalten nicht. Daher schneiden diese Systeme üblicherweise bei Angriffen, die auf Missbrauch von Zugangsdaten aufbauen, nicht gut ab. Die Abwehr solcher Angriffe benötigt einen grundlegend anderen Ansatz. Die Systeme müssen Sequenzen von HTTP-Anfragen untersuchen, anstatt sich auf einzelne Anfragen zu fokussieren. Des Weiteren nahm die von APIs verarbeitete Datenmenge zu, wodurch Machine Learning jüngst an Popularität gewann.

In der vorliegenden Arbeit präsentieren wir ein neuartiges System zur Erkennung von Anomalien im Benutzerverhalten. Unser Ansatz basiert auf einem Deep Neural Network (DNN), welches als Autoencoder konstruiert wird. Das vorgeschlagene System erkennt Anomalien anhand des Rekonstruktionsfehlers. Existierende Arbeiten analysieren üblicherweise die gesamte HTTP-Anfrage. Unser System verwendet hingegen lediglich Logdaten, welche nur einen Bruchteil der verfügbaren Informationen beinhalten. Unser Ansatz benötigt weder Daten vom Computer des Benutzers, noch benötigt er vorab eine Liste aller möglichen HTTP-Anfragen.

Zur Evaluierung unseres Systems verwenden wir Logdaten eines Online Stores. Die Ergebnisse untermauern die Annahme, dass die verwendeten Logdateien ausreichend Informationen über Verhaltensmuster beinhalten. Außerdem bestätigen wir, dass der Ansatz, Informationen vor der Verhaltensmodellierung zu komprimieren, auch mit Logdaten funktioniert. Abschließend zeigen wir, dass ein Kompromiss zwischen *dem Auffinden aller Angriffe (Recall)* und *dem Vermeiden zu vieler Fehlalarme (Precision)* gefunden werden muss.

Contents

List of Figures

List of Tables

List of Acronyms

Adam Adaptive Moment Estimation. 19, 20, 46
AI Artificial Intelligence. 9, 10, 77
API Application Programming Interface. v, vi, 1, 27, 61, 84
AUC Area Under Curve. 71, 74, 81, 82, 85

BPTT Backpropagation Through Time. 26

CAE Contractive Autoencoder. 29
CLI Command Line Interface. 60
CNN Convolutional Neural Network. 24, 28, 32, 33, 78, 85
CPU Computer Processing Unit. 60

DAE Denoising Autoencoder. 29
DNN Deep Neural Network. v, vi, 3, 7, 35, 84
DVAE Disentangled Variational Autoencoder. 30

FN False Negative. 4, 69
FNR False Negative Rate. 71
FP False Positive. 4, 69
FPR False Positive Rate. 70, 71, 80

GAN Generative Adversarial Network. 30
GBT Gradient Boosted Tree. 7
GCP Google Cloud Platform. 61, 63
GPU Graphical Processing Unit. 45, 60, 61
GRU Gated Recurrent Unit. 26, 27, 32, 33, 40, 42, 43, 78, 80, 85

HIDS Host-Based Intrusion Detection System. 7
HMM Hidden Markov Model. 7, 10, 80, 81
HTTP Hypertext Transfer Protocol. v, vi, 2, 3, 24, 27, 29, 30, 32, 35, 39, 50–52, 54, 80–82, 84

IDS Intrusion Detection System. 2, 6–8, 32
IPS Intrusion Prevention System. 2, 7, 32

LSTM Long Short-Term Memory. 21, 26, 27, 32, 33, 78, 80–82

MAE Mean Absolute Error. 19
MLP Multilayer Perceptron. 23
MSE Mean Squared Error. 19, 29

NAT Network Address Translation. 52
NIDS Network-Based Intrusion Detection System. 7
NLP Natural Language Processing. 25

PBT Population-Based Training. 16
PCA Principal Component Analysis. 10, 27, 28
PPV Positive Predictive Value. 70

ReLU Rectified Linear Units. 18, 46
ResNet Residual Network. 21
REST Representational State Transfer. 27
RNN Recurrent Neural Network. 26, 28, 31–33
ROC Receiver Operating Characteristic. 71, 73, 74, 81, 82, 85

SAE Sparse Autoencoder. 29, 42
SGD Stochastic Gradient Descent. 19, 20, 46
SOC Security Operations Center. 2
SQL Structured Query Language. 2
SVM Support Vector Machine. 10

TCN Temporal Convolutional Network. 32, 33, 78, 85
TN True Negative. 69
TNR True Negative Rate. 70
TP True Positive. 69
TPR True Positive Rate. 70, 71, 80
TPU Tensor Processing Unit. 60

VAE Variational Autoencoder. 30

WAF Web Application Firewall. 2

XSS Cross-Site Scripting. 2

1. Introduction

Modern software for consumers, businesses, and industrial areas increasingly relies on Application Programming Interfaces (APIs) as a central access point to data. In practice, the capabilities of APIs vary widely. On the one hand, basic read-only APIs providing data from sensors or surveillance cameras exist. On the other hand, there exist APIs that allow controlling devices such as engines, pumps, or smart home devices.

As APIs can provide access to secure environments, they pose a valuable target for attackers. For instance, APIs act as a data hub for consumer devices, server backends, or industrial edge devices. Therefore, attacks on APIs may cause damage to machinery behind it or theft of sensitive data, including private customer information or business secrets. In rare cases, attacks might even concern public safety. In a global context, APIs typically allow arbitrary connections from the Internet. Publicly exposing them requires advanced security mechanisms to prevent both external and internal attacks.

Securely operating an API requires precautions in numerous aspects. One crucial point is the close monitoring of the state of the API. The main goal is to block malicious requests that exploit unknown vulnerabilities. Doing so, can prevent damage caused by adversaries on the one hand, and provide hints about errors in the code on the other hand. Ideally, rejecting malicious requests takes place early in the request handling pipeline.

However, correctly classifying requests as benign or malicious is challenging. Two major factors make anomaly detection difficult. First, anomalies are sometimes subtle and hard to spot. Second, modern anomaly detection systems examine enormous amounts of data.

Research concerning the topic was published continuously throughout the last three decades. Denning [Den87] presented one of the first anomaly-based approaches towards intrusion detection. In the early days, Intrusion Detection Systems (IDSs) ran separately from the actual system. The goal was to identify possible incidents retrospectively. Nowadays, systems are capable of identifying anomalies in real-time. This way, the systems can prevent the execution of malicious requests.

Modern state-of-the-art monitoring systems either rely on network packet information or application data to identify dubious requests. Systems utilizing network packets can be classified as Intrusion Prevention Systems (IPSs), systems using application data as Web Application Firewalls (WAFs). Depending on the kind of attack, either both, none, or one of the two systems will alert the Security Operations Center (SOC).

The state-of-the-art systems perform reasonably well in detecting attacks that need special characters to successfully mount the attack, like Structured Query Language (SQL) injections or Cross-Site Scripting (XSS) attacks. While rule-based systems achieve acceptable detection rates, recent research on the topic has emphasized the benefits of machine learning for detecting malicious requests. Mac et al. [Mac+18], for example, use a neural network to classify HTTP requests as benign or malicious.

The real-time requirement of current systems also entails that systems need to be able to find patterns in large amounts of data quickly. This is why research increasingly relies on machine learning approaches. Moreover, it can find relationships in data that humans are not capable of identifying.

1.1. Challenges of API Protection

Machine learning-based systems can detect relations in data that are difficult to represent with static rules. Even if so, maintaining such complex rules is not feasible in reality. Furthermore, typical software increased in size so much that the number of rules needed to secure a system is overwhelming.

Both arguments accelerated the shift towards machine learning-based techniques significantly. However, current systems neglect relationships between individual requests. They focus on a single request at a time only.

In this thesis, we deal with a new type of attack: Attacks based on credential or trust abuse. Typically, those attacks do not generate invalid requests and are thus not detected by conventional systems. Identifying such intrusions builds on the hypothesis that adversaries abusing credentials expose deviating usage characteristics. Deviations range from (a) sending requests in a different order, (b) sending requests a benign user does not send at all, to (c) sending the same requests, but with noticeably different parameter values. Describing such far-reaching combinatorial possibilities with conventional systems is not possible. Therefore, learning a usage model with machine learning techniques is a promising way to pursue further. Models can target the whole system, user groups, or individual users. The narrower the target, the more instances of the model must be trained, which requires significantly more resources.

1.2. Detecting Behavior Change with Log Files

We present a system that detects anomalous user behavior based on recorded user activity. In contrast to existing work, it uses less input information. A Deep Neural Network (DNN) decides whether a given request is part of the previously learned usage behavior model. For a solid decision, the network receives a sequence of recent requests as input. The system outputs the next request it expects based on the learned model. Whenever the prediction does not match the actual next request, the system classifies the request as an anomaly.

Our system uses standard HTTP log files. Log files contain only a fraction of information of a full HTTP request. Hence, POST bodies and HTTP headers are not available. Fields we used are the *source IP*, *request DateTime*, *HTTP method*, *full path*, *referer*, and the *user agent*. We evaluate our model with log files from an online store recorded by Zaker [Zak19].

We restrict our work to data available before the request handling has started. As a result, our system relies on significantly less information compared to other work. For example, Du et al. [Du+17] based their detection system on execution traces of the request handling pipeline. Execution paths of software expose more information than the data triggering the execution do. Hence, the limitation on pre-execution data can impact the detection rate negatively compared to approaches using an extended set.

The detection rate is the most important metric for evaluating an anomaly detection system. A good system must achieve low rates for both False Positives (FPs) and False Negatives (FNs) to gain acceptance. Therefore, anomaly detection requires a well-balanced trade-off between raising false alarms and missing an intrusion. We show that smartly selecting the parameters of the model influences the ratio of wrongly classified samples significantly.

To summarize, this work demonstrates a novel system that mines web server log files for usage patterns. After training, the system is capable of identifying divergences from the learned behavior. Our work proves that web server log files contain sufficient information to learn behavior patterns of users. By detecting changes in the behavior, it is possible to counteract emerging types of attacks such as credential abuse and insider threats.

1.3. Outline

This thesis is structured as follows:

In Chapter 2, we give a sound basis of technologies needed for our approach. We start with event monitoring and anomaly detection tools in general. Next, we introduce general machine learning concepts. The focus then shifts towards neural networks and the different architectural types available.

Following that general introduction to the topic, Chapter 3 presents our developed system. The chapter starts with the overall description of our system. It then describes the architectural big-picture of the neural network. Finally, we give the exact specifications of the layers used. We also reason about the design decisions made during the development phase.

Next, Chapter 4 introduces the dataset used for this work. Before doing so, the chapter explains the essential decision criteria for a dataset. The final section of the chapter explains in detail the preprocessing steps performed.

In Chapter 5, we shift the attention towards the performance of our detection system. First, we discuss the development process and the tools used. Next, we describe the training environment. We then explain the metrics used to evaluate the performance. Finally, we present the results delivered by our user behavior modeling systems.

Following our results, we present work related to anomaly detection in Chapter 6. The presented papers either use a similar dataset with a different approach or a similar approach but based on different input data. We emphasize smart design decisions, that are possibly beneficial for our approach, but also mention limitations if there are any.

Finally, Chapter 7 concludes this thesis. It summarizes the main aspects of this work and proposes interesting aspects that could be inspected in further research done in this area.

2. Background

This chapter gives an overview of the techniques needed for the presented approach. First, it introduces concepts used to detect malicious actions initiated from external locations or from within an organization or company. Next, we define the basic terms and approaches in the field of machine learning. Following the general concepts, the focus shifts towards neural network-specific techniques. The final section of the chapter deals with the different architecture types and explains their strengths and weaknesses. Also, it gives reasons why the architecture type might be beneficial for the presented approach. In the following, we discuss different aspects and approaches to monitoring events related to security aspects.

2.1. Monitoring Security Relevant Events

Modern organizations need to observe the actions of internal or external actors thoroughly. There exist a variety of tools to achieve this task. For instance, companies rely, amongst others, on IDS to prevent external access to resources. Preventing internal actors like employees from stealing information usually focuses on passive actions like strictly configured access permissions. Nevertheless, active defenses like creating a usage model gain popularity. This section explains aspects of intrusion detection and insider threat detection. Since both approaches aim to detect outlying behavior, the section finally covers the basics of anomaly detection.

2.1.1. Intrusion Detection and Prevention

Intrusion Detection Systems (IDSs) monitor the events happening on a device or in a network and identify bad behavior. Intrusion prevention forms an extension of intrusion detection. Despite detecting the intrusion, *Intrusion Prevention Systems (IPSs)* are also capable of stopping the intrusion. Systems implemented solely in software, as well as systems deployed onto separate hardware, exist for this task. In the aftermath of an incident, the information recorded by an IDS is crucial for identifying possible points-of-entry of the attackers and performing a fast and clean system recovery.

The literature [AH18; Vin+19] distinguishes between Network-Based Intrusion Detection Systems (NIDSs) and Host-Based Intrusion Detection Systems (HIDSs). The TCP/IP layer serves as a source to differentiate between the two classes. *NIDSs* use header information of the TCP packets. Systems also inspecting the payload exist, but the increasing usage of encryption renders it costly, if possible at all. Furthermore, inspecting the payload of real traffic raises privacy concerns. *HIDSs*, in contrast, operate on the application layer. On the application layer, all the data are available in plaintext and accessible by the IDS.

Liao et al. [Lia+13] define three major intrusion detection methodologies: *signature scanning*, *anomaly detection*, and *stateful protocol analysis*. While signature-based methods are often easier to implement, they cannot detect previously unseen intrusions. Learning profiles of regular usage is a challenging task. Early approaches use If-Then-Else rules. Recent approaches use techniques based on machine learning, such as DNNs [Vin+19], Gradient Boosted Trees (GBTs) [FD19], and Hidden Markov Models (HMMs) [CQL17].

Khraisat et al. [Khr+19] compare the strengths and weaknesses of anomaly-based approaches to signature-based ones. They conclude that signature-based approaches are very effective and produce very few false positives. In contrast, anomaly-based approaches can be falsely triggered by users with dynamic activity profiles too frequently. They further distinguish anomaly-based approaches that try to identify anomalies in individual requests from approaches that identify anomalies in a stream of requests. The separation conforms with Kenkre, Pai, and Colaco [KPC15], who named approaches targeting individual requests *anomaly-based model IDS*, and models targeting request sequences *misuse-based model IDS*. The work we present is of the latter type, as we aim to detect changes in the behavior of users.

2.1.2. Insider Threat Detection

Organizations and companies face the risk of information being stolen and leaked by employees. Tuor et al. [Tuo+17] define the term *insider threat* as "any actions taken by an employee which are potentially harmful to the organization." Employees enjoy an elevated level of trust within a company. It is a noteworthy overhead for organizations to track which employees have access to which resources. However, it reduces the risk of insider threats significantly. Due to the elevated trust, insiders bypass some layers of the defense system like intrusion detection, firewalls, and have in-depth knowledge of the internal structure. Yuan et al. [Yua+18] point out the variety of attacks an insider might perform, like placing logic bombs or stealing intellectual property. Furthermore, they argue that insiders can hide their activity well, as most insider-based attacks happen during working hours where the majority of events happen.

Recent approaches use machine learning to identify insider threats. Machine learning performs well in detecting patterns in large amounts of data. Also, machine learning has advantages for complex decision problems where the transition from *class A* to *class B* is blurry. Events of users serve as a basis for normal user behavior, and divergences from that expected behavior raise alarms. The types of events used, as well as the preprocessing steps performed, vary a lot regarding different approaches.

2.1.3. Anomaly Detection

Anomaly detection describes the problem of finding patterns in a set of data points and reporting entities not following the identified pattern. These unexpected points are called anomalies or outliers. The source of the data on which to perform anomaly detection is not restricted in any kind. Therefore, Chandola, Banerjee, and Kumar [CBK09] list fraud detection of insurances or credit cards, intrusion detection, fault detection of critical systems, and military use cases as possible fields of application for anomaly detection. Gao et al. [Gao+18] further define the term *anomaly* to describe one of three different kinds of anomalies. The first kind is an unexpected change in a time series of data. The data often origin from sensors attached to mechanical devices. While the device is functioning correctly, the difference to the preceding value, as well as the absolute value, are in well-defined ranges. Predictive maintenance applications build upon such outliers. Depending on the data source, the time series might fluctuate due to changes in the environment - for example, day and night differences, or seasonal changes.

The second kind of anomaly identified by Gao et al. [Gao+18] is the difference of one entity compared to the other entities. Entities are clustered to detect anomalies. Entities not belonging to any of the identified clusters are considered anomalies.

Finally, the authors identify a third kind of anomaly, where the context influences the anomaly decision as well. The entity itself might be valid, but when including environment variables as well, the entity might represent an anomalous instance.

For example, this could mean to mine a behavior invariant of a given user group. Once the system has learned the invariant, it validates whether new requests are valid according to the learned model. Events identified as being anomalous have to be classified whether they are malicious, too. Kreimel, Eigner, and Tavolato [KET17] use machine learning to classify anomalies and decide whether it is anomalous.

2.2. Introduction to Machine Learning

Machine learning is a subfield of Artificial Intelligence (AI) that builds on training data. Chollet [Cho18] defines machine learning as a process of searching for useful representations in vast amounts of data. The exact learning technique used depends on the format of the data available for the training. A crucial aspect of algorithm selection is whether the data are labeled. Labeled datasets contain the target output value. The performance of a machine learning algorithm is strongly related to the quality of the data used for training. Therefore, to create well-performing models as well as to speed up the training process, the data are preprocessed. The detailed steps highly depend on the dataset in use, but García, Luengo, and Herrera [GLH15] list data transformation, data cleaning and noise removal, data reduction, and data integration from multiple sources as significant aspects of preprocessing.

2.2.1. Learning Techniques

Goodfellow, Bengio, and Courville [GBC16] divide the learning techniques into three different categories: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning. Supervised learning techniques require target values in the dataset. The target value can be either a class, in which case the process is called classification, or it might be a continuous value, then the process is called regression. Standard algorithms used for supervised learning are Support Vector Machine (SVM) [CV95], HMM [RJ86], and naïve Bayes [RN95].

Unsupervised learning. Unsupervised learning can gain information from datasets without getting feedback on the current performance. Feedback is not possible because the target values are unknown. García, Luengo, and Herrera [GLH15] describe unsupervised learning as finding similarities, relationships, and regularities in the input data. The approaches are often cluster-based. Well-discussed algorithms from this class are k-means clustering and Principal Component Analysis (PCA) [GBC16].

Reinforcement learning. Reinforcement learning [SB98] focuses on the optimization of exploit-or-explore situations. Teaching an AI to play a game is an exemplary use case. The algorithm has to find a strategy to decide how much of its capacity it should invest to use existing knowledge and collect some reward versus how much it should focus on improving the knowledge.

In the context of our work, we focused on unsupervised learning approaches. Log files are a well-established data source for anomaly detection and do not contain information whether the request was anomalous. Labeling the datasets by hand is not feasible. Also, it is an error-prone task, as malicious attacks are often subtle. Using software for this task does not guarantee that all samples are labeled correctly.

An additional important classification of training algorithms performed in literature is into online and offline trainable ones. If an algorithm is online trainable, the learned model can extract information from newly available data without the need for a full re-training. So these models can react to changes faster.

2.2.2. Data Preprocessing

The term *data preprocessing* is generally understood to cover all steps needed to transform raw data into a format suitable for machine learning techniques. García, Luengo, and Herrera [GLH15] summarize the steps to be performed as (1) data cleaning, (2) data transformation, (3) data integration, (4) data normalization, (5) missing data imputation, and (6) noise identification. Additionally, they argue that data reduction is part of preprocessing, too. Standard techniques for data reduction are feature selection, instance selection, and discretization. In the following, we will explain the preprocessing steps used in our approach in detail.

Discretization describes one possible way of data transformation where numerical attributes are replaced with nominal attributes. Additionally, during data transformation, the representation of classes is adjusted to represent the so-called one-hot encoding. A vector of length N is required to represent N classes. Each feature in the encoding vector represents one class, as shown in Table 2.1. For categories with large numbers of possible values, one-hot encodings waste much memory as it will be filled sparsely only.

Text-based data require data transformation, too. A common way to represent textual data is to use an *embedding*. An embedding is either word-based or character-based. Embeddings consider the context in which words are used and place them into a vector space. The algorithm tries to arrange the words in a way that words with a similar meaning are neighbors in the generated vector space. However, creating a well-performing embedding is a computationally intensive task. Therefore, pre-trained embeddings like Word2Vec [Mik+13], FastText [Jou+17; Boj+17], and GloVec [PSM14] are commonly re-used. Popular dimension sizes for embeddings covering a human language range from 50 to 300.

Preprocessing numerical values increases the performance of the learning algorithm. Goodfellow, Bengio, and Courville [GBC16] define normalization and standardization as common preprocessing steps for continuous data. *Normalizing* data means to shift the range of possible values to $[0;1]$. *Standardization* ensures that the mean μ of the data is 0, and the standard deviation σ is 1. If standardization or normalization should be preferred cannot be said in general. It depends on the underlying data.

During preprocessing, the dataset is split into three parts. The sets are called test, train, and validation sets. The *training set* acts as the actual information source from which the algorithm learns. The *validation set* is used during training to ensure the algorithm learns general and abstract concepts hidden in the data and not to reconstruct the input data. The *test set*, finally, is used once the training is completed to assess the performance of the machine learning algorithm on new and previously unseen data.

The data used in the individual sets must not overlap. The set size ratio is not strictly defined and depends on the overall amount of data available for the learning process. Common split ratios of the test and train set, as well as the train and validation set, are between 0.25 and 0.33.

In some cases, the dataset is too small for the splitting mentioned above. The *k-fold cross-validation* [GLH15] tries to solve this issue by splitting the data into k batches of equal size. During each iteration, a different batch takes the role of the validation set.

	color [red, yellow, blue]	size [S, M, L]
0	[0, 1, 0]	[1, 0, 0]
1	[1, 0, 0]	[0, 0, 1]
2	[0, 0, 1]	[0, 1, 0]
3	[0, 0, 1]	[1, 0, 0]

Table 2.1.: Example of four samples with two features named *color* and *size*, both one-hot encoded.

2.3. General Machine Learning Concepts

The field of machine learning uses a variety of terms and concepts when talking about a model's performance and its architecture. Some of the terms are specific to certain learning algorithms, and others are independent of the concrete approach selected. In this section, we focus only on general terms, used throughout the field. This section defines a common terminology that is used throughout this document.

2.3.1. Overfitting, Underfitting and Model Capacity

When comparing different machine learning models, the performance on previously unseen data is more significant than the performance on the training data. Machine learning algorithms define a cost function that is used to determine the performance. The *training error* relates to the cost function being applied to the data known during training. Goodfellow, Bengio, and Courville [GBC16] define the task of minimizing the training error as an optimization problem. The challenge in machine learning is to find abstract features in the training dataset. By using the identified abstract features, the machine can then handle previously unseen data. We henceforth refer to the machine learning algorithm's ability to generalize as *test error*. It is a direct indicator of the machine's performance. Therefore, it is an important metric when comparing different techniques.

Another metric is the *model capacity*. It influences the performance of any machine learning algorithm significantly. Factors that define the model

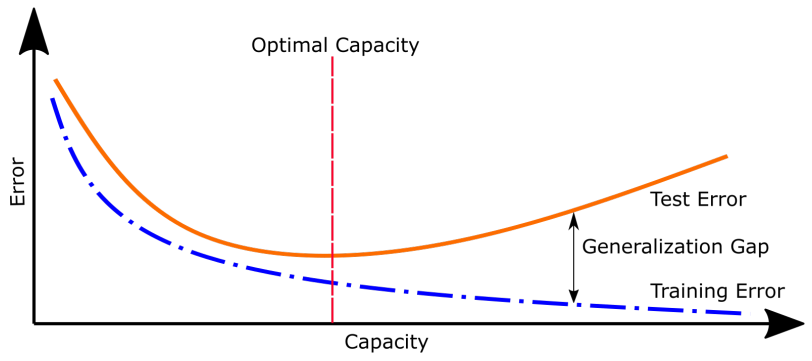

Figure 2.1.: Illustration of optimal capacity, overfitting, and underfitting. Adapted from figure 5.3 by Goodfellow, Bengio, and Courville [GBC16].

capacity depend on the precise algorithm used. For linear regression, increasing the capacity means using a polynomial of higher degree. For neural networks, it means to use more hidden units or hidden layers. If the selected capacity is too low for the problem to be solved, the model will *underfit*. An underfit model is not able to learn the abstract features in the data because the underlying cost function is too complex. Underfitted models perform poorly on the train set as well as on the test set. A model that remembers the samples it has seen instead of performing a meaningful generalization is *overfitting*. When the test error no longer decreases but instead starts to rise again, this indicates that overfitting starts. Figure 2.1 shows the concepts of over- and underfitting.

2.3.2. Early Stopping

Early stopping [GD98; SKP97] is one of the first approaches proposed to avoid overfitting. When the model starts to overfit, the training error continues to decrease while the test error starts to rise again. At this point, early stopping suggests not to continue with the training. However, the training should not be stopped immediately when the error increases. Instead, early stopping must tolerate occasional spikes in the validation error. The mechanism should step in action only if the validation error does not improve for several epochs. The generalization gap is likely to increase in later epochs as observable in Figure 2.1. A variant of early stopping remembers the parameter value of the epoch with the lowest validation error. The system trains for the predefined number of epochs independent of how the generalization error evolves. During the training, the system stores the parameters of the best-performing epoch and uses that parameter set at the end.

2.3.3. Hyperparameter Search

Hyperparameters and model parameters determine the performance of a model. In contrast to model parameters, the hyperparameters are not part of the model but considered external.

Patterson and Gibson [PG17] define *hyperparameters* as being responsible for the overall capability of a model to generalize. Additionally, they influence the time needed for the learning process. The set of hyperparameters is not exclusive and depends on the specific machine learning algorithm used. Examples of parameters that apply to a broad set of machine learning approaches are learning rate and model architecture [PG17; GBC16; Cho18], regularization [PG17; Cho18], epochs [Cho18], and momentum [PG17]. The hyperparameters are initialized with values chosen by humans. By adjusting the hyperparameters during the training, the system can speed up the learning process or escape a local minimum. However, according to Goodfellow, Bengio, and Courville [GBC16], hyperparameters that influence the model's capacity should not be optimized because this results in overfitting.

To find the optimal set of hyperparameters, one has to test every combination possible. Iterating all combinations possible is called a *grid search*. However, a grid search becomes infeasible quickly as the number of possible combinations explodes. Several strategies like random search [BB12], Bayesian optimization [Sha+16], and population-based approaches [Jad+17] aim to speed up the parameter selection process.

The basic idea behind the *random search* approach is not to waste too much time on poorly performing parameter values. Instead of changing only one parameter value per test run, Bergstra and Bengio [BB12] suggest changing all parameter values randomly. Doing so increases the chance to find a well enough performing parameter set, although it might not represent the best

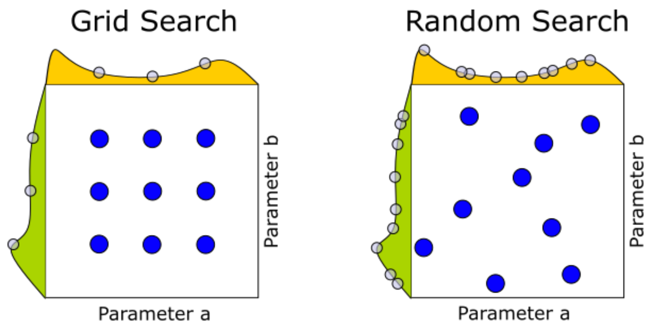

Figure 2.2.: Comparison of grid search and random search to find hyperparameters. Adapted from figure 1 by Bergstra and Bengio [BB12].

performing one. Figure 2.2 illustrates the differences between grid search and random search.

Both grid search and random search do not use information from previous runs when selecting the next set of parameters to test. Shahriari et al. [Sha+16] try to tackle the mentioned problem with *Bayesian optimization*. For each parameter, they create a model which outputs the next value to try based on the previous performances.

The strategies mentioned above all either assume sequential training runs or do not interconnect training runs executed in parallel. Jaderberg et al. [Jad+17] proposed *Population-Based Training (PBT)*, which was inspired by the evolution theory. At first, they start several different models concurrently. The system monitors the performance of the individual instances closely. If the system detects a poorly performing instance, it aborts the training of this instance and replaces it with the current weights of a well-performing one. However, the training continues with the original set of hyperparameters. If the instance fails repeatedly, the system kills it. A mutant of a well-performing instance will take the free spot.

Generally speaking, applying PBT is a challenging task as one must have the resources to run several instances in parallel. Additionally, a framework that monitors the training progress and takes appropriate actions if needed has to be developed. Developing the model is an iterative process. A possible approach one could take is to test parameters at random in the beginning. This step aims to define well-performing ranges for the parameters. After that, the fine-grained parameter search can follow a more structured approach.

2.4. Neural Network Basics

The machine learning concepts presented so far apply to the majority of techniques available. This section focuses on neural network specifics. Neural networks can handle many different domains with varying input formats like image data, audio streams, videos, text, time series, and numeric data. We use neural networks because they can handle text input as well as sequential information. Both aspects are essential for log-based anomaly detection. First, this section explains the core components needed to build a neural network. Then, we present optimization techniques to improve the performance and reduce the training time. Techniques suitable to prevent a model from overfitting conclude the section.

2.4.1. Layers

Neural networks are inspired by the neurons in a humans' brain. A neural network consists of at least one input layer and one output layer. In between, the network can have hidden layers. Literature does not define whether the output layer counts towards the number of layers of a network. The input layer, however, never counts towards the total number of layers. Kröse and Smagt [KS96] defends the convention because the input layer does not perform any calculations. For clarity, through this thesis, we refer to a network composed of one input layer, one output layer, and one hidden layer in between as having one hidden layer.

Each layer consists of *perceptrons*. Neuron or node are terms often used as an alternative for perceptron. Each perceptron has connections to the neurons of the previous and next layer. Equations 2.1 and 2.2 show the computations performed by each perceptron.

$$z^{(l)} = W^{(l-1)} * a^{(l-1)} + b^{(l-1)} \tag{2.1}$$

$$a^{(l)} = f(z^{(l)}) \tag{2.2}$$

Equation 2.1 defines the calculation of the interim value z. Each connection of a neuron in layer $l-1$ to a neuron in layer l has an assigned weight. The weight matrix W represents those weights. The bias vector b represents the biases of the neurons. Equation 2.2 shows the calculation of the final value a. It uses a non-linear activation function f with the result of the first equation as input. Finding the right values for the weights and the biases is the main challenge of the training process.

2.4.2. Activation Functions

Activation functions are part of neurons and assigned layer-wise. This assignment ensures that all neurons on the same layer use the same activation functions. According to Chollet [Cho18], the activation function contributes a non-linearity to the layer. Without non-linearity, the layer could perform linear transformations only, and stacking multiple layers would not be beneficial.

For hidden layers, a commonly chosen activation function is a *sigmoid*, like the logistic function $f = \frac{1}{1+e^{-x}}$ or the hyperbolic tangent $f = tanh(x)$. LeCun et al. [LeC+98] recommend using *tanh*, because it centers around the origin and is symmetric, which results in better performance. *Rectified Linear Units (ReLU)* [NH10] is another popular activation function and is defined as $f = max(0, x)$. The popularity comes from its cheap computational costs and fast convergence. Figure 2.3 shows the plots of the described activation functions.

The *softmax* function is a popular activation function for the output layer and is usually used for multi-class classification tasks. The softmax transforms its input to a probability distribution. Its inputs are usually unscaled, which means they can sum up to any value in the interval $[-\infty; +\infty]$. The unscaled inputs are called *logits*. Goodfellow, Bengio, and Courville [GBC16] show details as well as actions necessary for numeric stability.

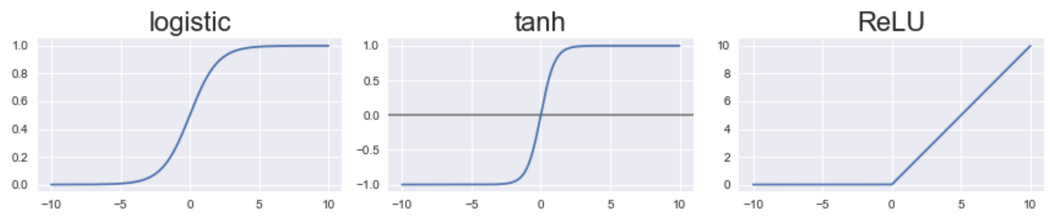

Figure 2.3.: Comparison of activation functions: logistic, tanh, and ReLU.

2.4.3. Loss Functions

Loss functions, often also called cost functions or error functions, measure the differences of the output of a model to the target value. The training aims to minimize the loss function, which means to reduce the difference between the output and the target. The choice of the loss function is, therefore, crucial for a well-performing network. Depending on the task performed, different options for loss functions are suitable. Rosasco et al. [Ros+04] and Goodfellow, Bengio, and Courville [GBC16] list and explain several loss functions like Mean Squared Error (MSE), Mean Absolute Error (MAE) and Cross-Entropy loss. In summary, MSE and MAE perform well on regression problems. MAE is more robust against outliers than MSE is. The Cross-Entropy suits classification problems and exists in binary and multi-class flavors.

2.4.4. Optimizers

Optimizers define how to reduce the loss of the model, which eventually improves the quality of the output. Current state-of-the-art optimizers for neural networks rely on the minimization of the gradient of the loss function. Ruder [Rud16] gives an overview of popular optimizers and identifies Stochastic Gradient Descent (SGD), RMSprob, and Adaptive Moment Estimation (Adam) as the essential ones.

The *gradient descent* computes the gradient based on the whole dataset. For large datasets, this is not feasible as the time needed for just one parameter update is too large. *SGD*, in contrast, updates the weights after each sample, which can cause many fluctuations as not all updates tend to decrease the global error. The mini-batch-based approach splits the dataset into small batches of 32 to 256 samples each. Values outside the specified range are possible, but usually do not perform well. The best-performing batch size depends on the underlying dataset and has to be found during the hyper-parameter search. The parameter update happens after each mini-batch. *Mini-batch SGD* resembles a trade-off between parameter update accuracy and time needed for each update. Patterson and Gibson [PG17] argue that the optimal batch size balances memory requirements, computational efficiency, and optimization efficiency. A disadvantage of mini-batch updates is that it is not guaranteed to find the global minimum. Additionally, they suggest using batch sizes that are a power of two as these batch sizes utilize the hardware optimally.

Adam [KB14] is very popular as it tries to solve several drawbacks of SGD. It uses information from previous parameter updates to avoid oscillations. Also, it adjusts the learning rate for each parameter to speed up the optimization process. Adam performs better than SGD in many cases, but it is not guaranteed.

2.4.5. Back-Propagation

Rumelhart, Hinton, and Williams [RHW86] introduced the *back-propagation algorithm*. It is an efficient way to compute the gradient ∇ of a function. The back-propagation algorithm plays a central role during the training of a neural network. It aims to minimize the error of the loss function by updating the weights and biases of the neurons. The updates are calculated by applying the chain rule, which the authors explain in detail in the original paper. The learning rate ϵ regulates the update process and prevents the algorithm from jumping over the minimum. Choosing the right value for the learning rate is one of the main problems.

2.4.6. Deep Learning

Neural networks are often put into classes depending on the number of hidden layers they have. Networks with many hidden layers are called deep neural networks, networks with a few layers are called shallow. There is no sharp boundary defined when to consider a network as deep. However, GoogLeNet [Sze+15], an image classification network, uses 23 layers and is agreed on to be a deep network.

Despite the architectural differences, deep networks and shallow networks have different conceptual approaches too. Shallow networks require preprocessed data with already extracted features. Deep networks usually use the earlier layers to perform the feature extraction from raw data. The later layers learn generalized features based on the output of the earlier layers. Lecun, Bengio, and Hinton [LBH15] claim that the automatic feature extraction is the key advantage of deep learning approaches. However, basic data preprocessing like normalization or standardization is still advisable for deep neural networks for faster training.

2.4.7. From Vanishing and Exploding Gradients

During the training of deep networks, the back-propagation algorithm requires many multiplications due to the chain rule. Multiplications can cause problems if the values become very large or very low. For instance, if the values are below zero, the multiplications decrease the value even further. The literature refers to this problem as the *vanishing gradient problem* [Hoc91]. Solutions to the mentioned problem suggest architectural changes.

If the data are sequential, Long Short-Term Memory (LSTM) cells [HS97] provide a possible alternative. Our approach uses such cells for that very reason. The cells maintain an inner state, which helps to prevent the gradient from vanishing. He et al. [He+16] presented an architectural change called *Residual Network (ResNet)* that is applicable to any kind of input data. They added shortcut connections from layer N to layer $N + 2$, which adds the original weights to the outcome to prevent the gradient from vanishing. The two layers connected with the shortcut and the layer in between form a so-called residual block. Residual blocks can be stacked, too.

The opposite problem of values getting very large is called the *exploding gradient problem*. Large values often lead to overfitting. Pascanu, Mikolov, and Bengio [PMB13] suggested gradient clipping to prevent the network from overfitting. It works by defining a threshold that represents the maximum value the gradient may have.

2.4.8. Dropout

Dropout [Sri+14] is one of the most common techniques used to avoid overfitting. For each training run, a preconfigured percentage of neurons gets deactivated. A deactivated neuron does neither participate in the forward-passing of activations nor does it back-propagate the error. Without dropout, a neuron may learn to correct the mistakes of another neuron. If the error correcting neuron is not present all the time, the original neuron must learn a better abstraction of the data. Dropout is usually applied layer-wise. Using dropout introduces an additional hyperparameter called the dropout rate. According to the original paper, well-performing dropout rates range from 0.5 to 0.8. In literature, the dropout rate sometimes describes the number of neurons removed, and sometimes the number of neurons remaining. This thesis follows the scheme presented in the original paper, where a dropout rate of 1 is equivalent to no dropout at all.

2.4.9. Weight Regularization

Weight regularization is another technique used to avoid overfitting. Chollet [Cho18] argues that without weight regularization, the weights may become large and are not well distributed. A network with many large weights tends to react strongly to input changes. Adding penalties for large weight values encourages the network to use smaller weights. Goodfellow, Bengio, and Courville [GBC16] define the *L1* and *L2* vector norm as standard methods to determine the required weight penalties. The *L1-Norm* uses the absolute sum of weights, whereas the *L2-Norm* uses the squared sum. Squaring the values punishes large values even more.

2.5. Neural Network Architectures

There exist different neural network architectures for different tasks. Requirements to the neural network can be divergent, depending on the problem to be solved. For example, object detection in images should work independently of the location of the object within the picture. In contrast, the location of elements in a sequence is particularly important if the neural network should predict the succeeding value. In this section, we discuss the advantages of the different neural network architectures. Furthermore, we show how our approach can benefit from using the presented network architectures.

2.5.1. Multilayer Perceptron

The purest form of neural network found in literature is known as *Multilayer Perceptron (MLP)* [GD98; Hay05; Naz+08]. If no specialized network type is defined, the MLP is the default type. The network consists of an input, an output, and one or more hidden layers. Since one layer is connected to the previous and following layer only, a linear architecture without loops is defined. These networks are called *feed-forward* networks for that reason. Gardner and Dorling [GD98] define that each layer must have a non-linear activation function to be able to learn non-linear dependencies in the data. This kind of network is called *fully connected* because each node of layer N is connected to each node of layer $N + 1$. MLPs deliver satisfactory results for classification as well as regression problems. Turning to user behavior learning, we will use a more advanced base structure than an MLP. However, those advanced structures often include fully connected layers.

2.5.2. Convolutional Neural Network

Convolutional Neural Networks (CNNs) [LeC89] became popular with computer vision-related challenges. Khan et al. [Kha+19] refer to this field of research as *machine vision*. However, the concepts used by CNNs can be applied to one-dimensional data as well. The strength of CNNs lies in finding abstract features within the data. In computer vision terms, this means the network can detect features independent of the exact location within an image, its size, or its rotation. Liu et al. [Liu+17] define two types of layers called convolution layers and subsampling layers. The two types of layers form the heart of CNNs.

Convolution layers apply a kernel to the data. Typically kernel sizes range from $[2, 2]$ to $[5, 5]$. The kernel is applied in a sliding window approach with shared weights. It can detect features independent of their location in the data. Depending on the dataset, they can also detect rotated or resized features. Furthermore, sharing the parameters reduces the computational complexity. Chollet [Cho18] lists max-sampling and average-sampling as methods available for the *subsampling layers*. The subsampling layers again require a filter size, which defines the number of neighboring elements taken into consideration for the maximum or average computation. Convolutional and subsampling layers may be stacked multiple times. Subsequent layers then try to extract high-level features based on the outcome of earlier layers. Figure 2.4 illustrates an exemplary structure of a CNN

By default, both the convolutional and the subsampling layer, reduce the dimensionality of the data. Patterson and Gibson [PG17] show how to use padding to avoid the dimensionality reduction. The padding ensures that each of the original data cells resides in the middle of the kernel once. The padded values are either zeros or based on the neighborhood, like the average or the median.

Turning to our architecture, HTTP requests accept GET parameters in no particular order. The client implementation defines the order of the parameters. Applying CNNs on the query string can, in some instances, result in better performance. CNNs can identify two requests with the same parameter values passed but in a different order as being equal.

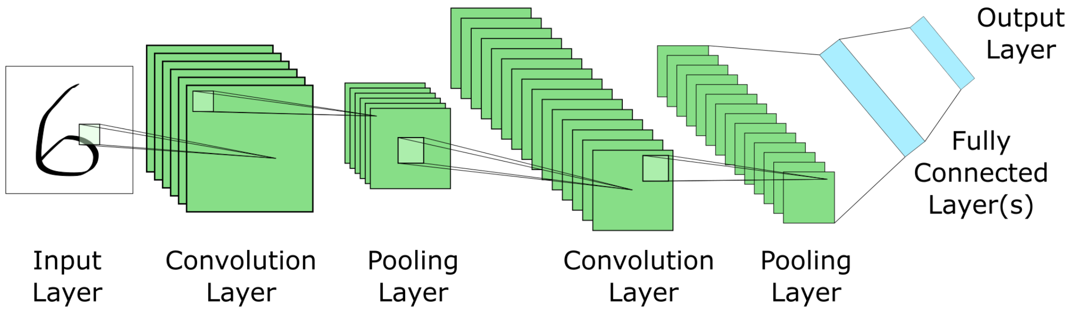

Figure 2.4.: Combination of convolution and subsampling layers as they are usually used in CNNs. The image is adapted from figure 5 by Liu et al. [Liu+17].

2.5.3. Sequence-Aware Architectures

Sequence-aware architectures perform well on tasks where the sequential information is essential. To remember long-term dependencies, they maintain an internal state depending on the elements seen earlier in the sequence. It is, therefore, popular for tasks, where the sequential information is of importance. Pouyanfar et al. [Pou+18] list Natural Language Processing (NLP) and speech processing as exemplary use cases for such architectures. During the computation, however, the network has to be unrolled, which makes it computationally expensive. The most basic architecture uses a forward computation only. This way, the context of the current sequence item depends on previous items, but not on the following ones. However, the following items might influence the context of a sequence item, like the meaning of a word, too. Schuster and Paliwal [SP97] suggested a bidirectional approach to tackle this problem. The bidirectional approach unrolls the network from the beginning and the end simultaneously. This way, information from the end of the sequence can contribute better to the model. In the following, we define three types of architectures that perform well regarding sequence modeling. Sequential features play a crucial role in our architecture as they act as a basis to learn user behavior.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) [RHW86] remember information over time by having a hidden state. They, however, perform better at modeling short-term dependencies than long-term dependencies. During training, the RNN is unrolled and the parameters shared across time. Despite reducing the number of parameters and subsequently reducing the total training complexity, the shared parameters strengthen the network's capability to deal with sequences of different lengths. Additionally, the shared parameters strengthen the network's robustness against minor variations in a sequence. An adaptation of the backpropagation algorithm called *Backpropagation Through Time (BPTT)* calculates the updates of the weights during training. When unrolling the network, it can become deep quickly. Deepness can cause problems like exploding and vanishing gradients.

Long Short-Term Memory Cells

Hochreiter and Schmidhuber [HS97] introduced *Long Short-Term Memory (LSTM)* cells to improve the models' capability to learn long-term dependencies in sequences. LSTM cells have an inner state and gates to modify that state. The gates are called *Input*, *Output*, and *Modify*. In Figure 2.5, we illustrated the gates of an LSTM cell and how they are connected to produce the output value. Over time, variants of LSTMs have been proposed that change the layout slightly. A noteworthy change suggested by Gers and Schmidhuber [GS00] is to use the cell state as input to the gate functions, too.

Gated Recurrent Units

Based on the great performance of LSTMs, Cho et al. [Cho+14] presented an improvement called *Gated Recurrent Unit (GRU)*. The authors merged the *Input* and *Modify* gates into a single gate called *Update*. By merging two gates, they reduced the number of parameters in the model. A smaller model size results in faster training and query times for the network. However, its learning performance sometimes cannot reach the performance of an LSTM.

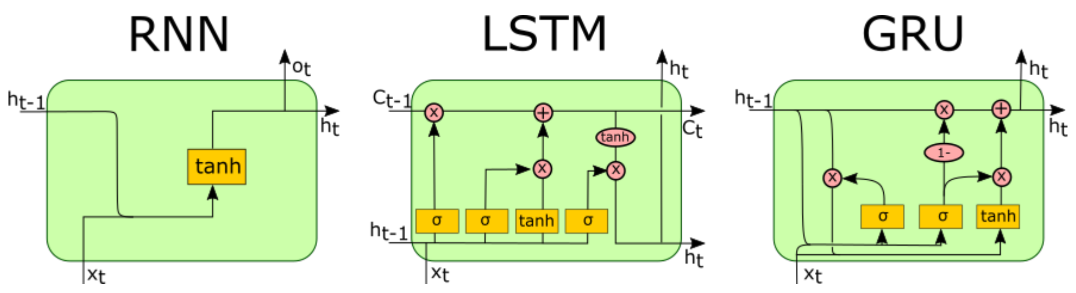

Figure 2.5.: Comparison of RNN, LSTM, and GRU cells. The image is based on illustrations of Olah [Ola15].

In their thorough survey, Alom et al. [Alo+19] concluded that neither LSTMs nor GRUs are universally superior over each other. GRUs are cheaper and train faster, LSTMs deliver better results if an extensive training set and computational power are available. Figure 2.5 compares the different types of nodes. Namely, the illustration contains regular recurrent nodes, LSTM cells, and GRU cells.

Focusing on HTTP requests, the API path often follows some design principles like Representational State Transfer (REST) [FT00]. GRUs and LSTMs can interpret the path information as a sequence of words or characters. If word-based or character-based approaches perform better cannot be said in general. Additionally, we use a sequence-aware structure to mine the user behavior. User behavior is composed of a sequence of events. In concrete terms, sequences of HTTP requests define the user behavior.

2.5.4. Encoder-Decoder Architectures

An *encoder-decoder architecture* describes a framework to construct a network that learns features of a dataset by itself. The encoding part of the network tries to extract useful features out of the input data. Following that, the decoder uses the extracted features and tries to map them into the desired output vector space. In many cases, the outcome of the encoder is of lower dimensionality than the original input. The compressed representation is often referred to as *latent space*. The dimension reduction forces the encoder to lean abstract features. Encoder-decoder architectures can be seen as a non-linear extension to Principal Component Analysis (PCA). Baldi and Hornik [BH89] show that, when using a linear encoder-decoder architecture, the learned weight matrix spans the same subspace than the results from a PCA. For non-linear encoder-decoder architectures, the output is only conceptually comparable to PCAs. The weights learned are not statistically independent components and not sorted by variability. Recent research [LNP19] tries to address this by starting with a small latent space and increasing it iteratively. Whenever the latent space increases, the model can learn a new dimension.

Reducing the input to a compressed set of abstract features decouples the information from the current form of representation. This way, encoder-decoder architectures can transform information from one form of representation into another like audio to text. Additionally, some encoder-decoder architectures can generate new instances not seen by the system before. Yan et al. [Yan+16a] combined those two properties and proposed an approach that generates images based on visual attributes. Exemplary attributes they defined for human face generation are *gender*, *eyes open*, *teeth visible*, and *pointy nose*.

General Encoder-Decoder

In general, encoder-decoder architectures do not require matching dimensions for input and output data. It is possible to use, for example, video or audio data as input and output text. Wu, Dinkel, and Yu [WDY19] use an audio stream as input and generate captions describing the contents of the stream. The different dimensions of the data usually lead to different architectures for the encoder and the decoder. Encoders and decoders can use concepts presented earlier, like CNNs and RNNs. The general encoder-decoder architecture requires a labeled dataset since the input and output vector spaces are different.

Autoencoder

Autoencoders are especially useful if no target data are available. The Autoencoder uses each sample as input and reference output. This implies that, in contrast to the general encoder-decoder architecture, the input and output vector space is the same. The reconstruction error measures the performance of an Autoencoder. A standard metric used to measure the *reconstruction error* is the MSE. The architecture incorporates a bottleneck that forces the network to learn abstract features. Also, it prevents the model from simply copying the input nodes. Goodfellow, Bengio, and Courville [GBC16] show a straightforward way to accomplish a bottleneck by using an undercomplete Autoencoder. An *undercomplete Autoencoder* has a latent space with fewer dimensions than the input layer. Alternative methods to force the network to learn abstract representations are called Contractive Autoencoder (CAE) [Rif+11], Sparse Autoencoder (SAE) [GBC16], and Denoising Autoencoder (DAE) [Ben+13]. The following paragraphs explain the methods and their characteristics.

Sparse Autoencoder. Sparse Autoencoders (SAEs) are overcomplete, meaning that they use a latent space larger than the input layer. A large number of features present in the data requires a larger latent space of the Autoencoder. Otherwise, it cannot focus on all features present. The system randomly deactivates some nodes in each run to prevent simple copying behavior. The disabled nodes force the network to use all the remaining nodes. SAEs can improve the performance of our model if the users expose many different behavioral features.

Denoising Autoencoder. Denoising Autoencoders (DAEs) learn to reconstruct the original data sample from slightly distorted samples. When using images as data samples, this means that a Gaussian noise is applied to the input images. The network again cannot simply copy the source data but has to extract useful features.

Contractive Autoencoder. Contractive Autoencoders (CAEs) aim to be less sensitive to small variations in the input data. In contrast to DAEs, which try to make the reconstruction function resistant against noise in the training data, the CAE makes the feature extraction function resistant against noise. For this reason, CAEs are popular as a feature extraction method for deeper layers. For outlier detection based on

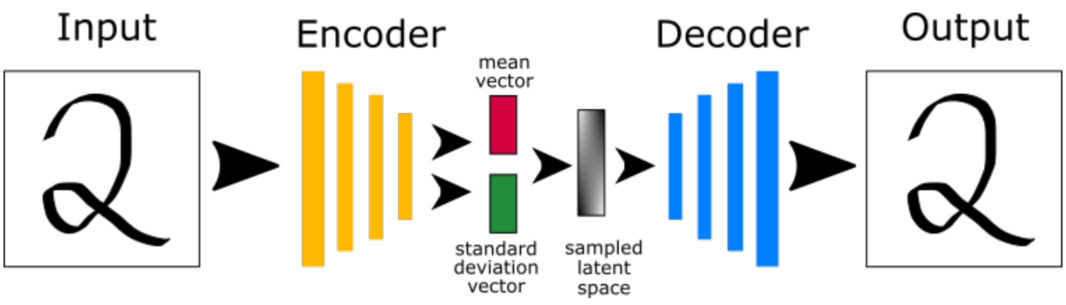

Figure 2.6.: Abstract structure of a VAE. The illustration is based on figure 9 of Weng [Wen18].

HTTP requests, CAEs can contribute by improving the quality of features extracted by the upper layers.

Variational Autoencoder

Variational Autoencoders (VAEs) [KW13; RMW14] not only learn abstract features but can produce entirely new samples. Standard Autoencoders are not enforcing any structure on the latent space, so taking a random vector and feeding it into the decoder can result in meaningless output. The VAE aims to learn μ and σ of a Gaussian distribution to avoid meaningless output. Figure 2.6 shows the layout of the architecture. The decoder accepts a sample from the learned distribution and bases its reconstruction process on that.

VAEs can be combined with *Generative Adversarial Networks (GANs)*, which are another type of sample-generating network. Larsen et al. [Lar+16] introduced the combination and called it VAE/GAN.

Without further enhancements of VAEs, it is impossible to control the characteristics of the generated samples. Li et al. [Li+19] presented *Disentangled Variational Autoencoders (DVAEs)*, where they ensure that each vector passed to the distribution is associated with one feature only. This way, more control of the sampled data point is possible. Applying this to our architecture would force the network to clearly separate features like *date*, *HTTP method*, and *path information* in the latent space.

2.5.5. Networks with Attention

Attention allows a neural network to decide which data points influence the current one. The attention mechanism originated from text translation tasks, where information located at the beginning of a sentence may influence the meaning of a word at the end of the sentence. Bahdanau, Cho, and Bengio [BCB15] proposed the attention mechanism to counter-fight the problem of long-term dependencies. The attention mechanism determines the influence of the other items in a sequence to the current one. Masking can hide elements at specific steps from the algorithm. Doing so forces the algorithm to focus on elements located before the current one only, for example. The excellent performance of the attention mechanism led to its application in other problem domains, too. For example, Xu et al. [Xu+15] showed that attention mechanisms could be used for image caption generation as well. A subfield that emerged from attention is *attention visualization*. By visualizing the area the neural network gives attention to, it is easier for humans to understand why a neural network gives some specific output.

Shifting the focus towards behavior detection, attention can support the encoding layers to find an improved compressed representation of the requests. The network can then decide which areas of a sequence are of importance and which are not.

2.5.6. Transformer

Transformers were proposed by Vaswani et al. [Vas+17] and represent an evolution of attention-based RNNs. A transformer is a combination of Autoencoders and attention mechanisms. It performs sequence-to-sequence modeling tasks. The original problem domain transformers tried to tackle was text translation. In contrast to the attention-based architecture proposed by Bahdanau, Cho, and Bengio [BCB15], the transformer does not need recurrent elements while still performing well at remembering long-term dependencies in sequences. Transformers also make use of positional encoding. The positional encoding ensures that the position of the word makes a difference. In contrast to RNN-based architectures, transformers are well-suited for parallel processing. Parallel processing reduces the time needed for training significantly. However, in contrast to other recurrent-based architectures like RNNs, LSTMs, or GRUs, the transformer is based on a fixed sequence length, which requires padding or truncating of sequences.

The main advantage of transformers is its suitability for parallel processing. Parallel processing reduces the training duration significantly. However, our work does not focus on improving the training performance but on the outlier detection quality. Therefore, we plan to use transformers only if they improve the overall detection performance.

2.5.7. Temporal Convolutional Networks

Temporal Convolutional Networks (TCNs) [BKK18] aim to replace the unfolding-over-time aspect, like transformers, too. TCNs are inspired by the human brain. The human brain remembers long sequences by splitting them into shorter sequences. Likewise, the brain does not process text word-by-word. TCNs adapt this mechanism. They use hierarchical attention-based encoding to learn a representation of a sequence. TCNs use the same convolution layers as CNNs. When combining them with an encoder-decoder architecture, the TCN acts as the encoder. Thus, the outcome represents the latent space. The TCNs utilizes the hardware much better. It accesses the data at once and not in sequential order

In theory, TCNs can improve the feature extraction of HTTP requests. As the order of GET parameters is interchangeable, the convolution layers have the potential to identify those requests as semantically equivalent correctly.

2.6. Summary

In this chapter, we presented techniques for monitoring security-relevant events. While IDSs can detect intrusions, IPSs can additionally prevent them. Liao et al. [Lia+13] define anomaly detection as one viable method used by such systems. Khraisat et al. [Khr+19] conclude that systems based on anomaly detection, such as ours, are likely to produce false positives. In return, they can detect previously unknown attacks. Insider threat detection monitors the activities of persons within the organization, preventing them from abusing their elevated trust level.

Next, we introduced fundamental machine learning concepts. Machine learning allows computers to find relationships in large amounts of data. Supervised learning, unsupervised learning, and reinforcement learning are techniques applicable depending on the dataset. Our system uses unsupervised learning because entries in log files do not have labels identifying them as benign or malicious.

Following that, we explained the fundamentals of data preprocessing. Important techniques available are discretization, standardization, and normalization. Which technique to choose depends on the feature. Additionally, we discussed common representations, namely one-hot encodings, text-based embeddings, and char-based embeddings.

We then established a common terminology to compare machine learning models. First, we explained overfitting and underfitting. Both indicate a model that did not generalize well. Viable countermeasures are early stopping [GD98; SKP97], dropout [Sri+14], and weight regularization. Additionally, we gave a comprehensive introduction to loss functions, optimizers, and hyperparameter search.

The final section of the chapter presented different architecture types. CNNs [LeC89] are very popular for computer vision problems. However, they can also handle one-dimensional data. CNNs suit problems that require detecting features independent of their geometric translation. TCNs [BKK18] use elements of CNNs and model sequential information.

GRUs [Cho+14], LSTMs [HS97], and RNNs [RHW86] are capable of modeling sequential data. LSTMs handle long sequences better than RNNs because they use an internal state as memory. GRUs enhance LSTMs by exposing a lower memory footprint. Combining them with attention mechanisms [BCB15] allows the network to decide which areas in a sequence are important.

Finally, we turned to encoder-decoder structures. An Autoencoder learns a compressed representation and uses that to reconstruct the input sample. Autoencoders reconstruct samples seen during the training well while reconstructing unseen samples badly. Hence, the reconstruction error serves as a measure for classifying outliers.

The next chapter explains our anomaly detection system. After giving a big-picture overview of the developed system, we explain in detail the layers and architecture elements used for our architecture. Whenever there are multiple viable options, we will explain our decision.

3. Behavior Modeling System

In this chapter, we focus on the developed behavior anomaly detection system. We start by showing the general approach to anomalous user behavior mining in request logs. We also derive the requirements for feeding the requests into the detection system. The requirements impact the preprocessing pipeline significantly. We decided to use a DNN for our anomaly detection system based on a claim of Chalapathy and Chawla [CC19]. The authors argue that DNNs are superior to simpler neural networks when the amount of training data increases.

Next, we focus on the architecture of the neural network. We describe our two-stepped approach: First, we perform a dimensionality reduction of the requests. Next, we learn user behavior based on that representation.

Following the high-level overview, we deep-dive into the specific layers of the individual components of the Autoencoders. We explain the technical details like the activation function, number of nodes, and dropout rates. Not only do we present the final architecture, but we also discuss the reasons for our selection if there are feasible alternatives available.

Finally, the chapter focuses on the hyperparameters defining the model. We list the individual parameters, discuss the tested ranges, and also specify the best-performing value.

3.1. System Overview

The developed system learns user behavior based on logs from HTTP requests. The more requests the system inspects at once, the more details and interconnections it can detect. However, the longer the request sequences,

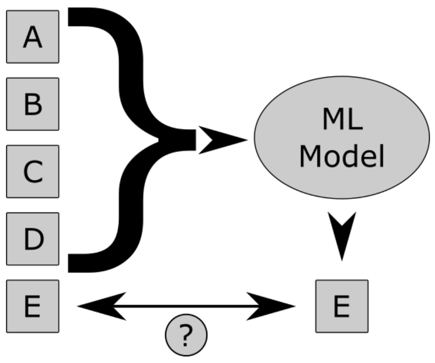

Figure 3.1.: High-level view of the developed system. It takes a sequence of samples as input, predicts the next expected sample, and compares it to the actual next request. The next request is anomalous if it does not match the predicted one.

the larger the computational power needed during the training of the neural network. Preprocessing ensures that the requests are in the correct order, split by user session. From the session request stream, a predefined number of requests serves as the initial input to the network. The input starts at the beginning of the event stream. Step by step, the window then slides over the remaining requests until it reaches the end of the request stream. We chose the sliding window approach because a deviation of behavior might happen at arbitrary points during a user session.

For each input request sequence, the model predicts the next expected request. If the current request does not match the predicted one, the system classifies it as an anomaly. Figure 3.1 illustrates the workflow. Our detection system uses a neural network based on Autoencoders to model this behavior. Autoencoders can cope with large amounts of unlabeled data, such as log files. As a result, we avoid using or generating a dataset with labels.

Neural networks, like most machine learning algorithms, require a numeric representation for text. Standard techniques are word-based and character-based tokenization. We chose the latter for our system. Character-based tokenization assigns an integer to each character. We ordered the characters by frequency. So the lower the integer, the more frequently the character appears in the data. The mapping of integers to characters is called *vocabulary*. The vocabulary size is smaller when using character-based tokenization compared to word-based tokenization. A smaller vocabulary increases the number of computationally feasible options available for our architecture. Additionally, character-based tokenization can represent words not seen during training. This is beneficial for parameter values where users can enter arbitrary text.

3.2. Neural Network Architecture

We start by describing the architecture on a conceptual level. We then continue with precise explanations of the Autoencoder structure, which consists of two distinct instances. In contrast to the big picture, these explanations include the layer details used for each Autoencoder. Layer details are, for example, the input and output dimensions, layer type, activation function, and connected previous and following layers.

3.2.1. Architectural Big Picture

At the heart of the developed architecture lies a two-stepped Autoencoder structure. The first Autoencoder, to which we refer as *request Autoencoder*, produces a reduced representation of the requests. It compresses each request of the sequence separately, not considering any information of neighboring requests. The sequence of reduced requests serves as input for the second Autoencoder, which we call *behavior Autoencoder*. In contrast to the request Autoencoder, it focuses on behavioral aspects. Figure 3.2 gives a high-level overview of the architecture.

The objective of the request Autoencoder is to find a compressed representation of the requests. Compressing the requests reduces the complexity of the behavior learning components building upon this representation. The Autoencoder must compress a request to the same latent space representation independent of its position within the sliding window. Hence, the Autoencoder compresses each request on its own to ensure location independence. Additionally, the request separation at this step prevents the encoding of any behavior information. However, decoupling the request compression layers contradicts the location independence criteria. To counteract that problem,

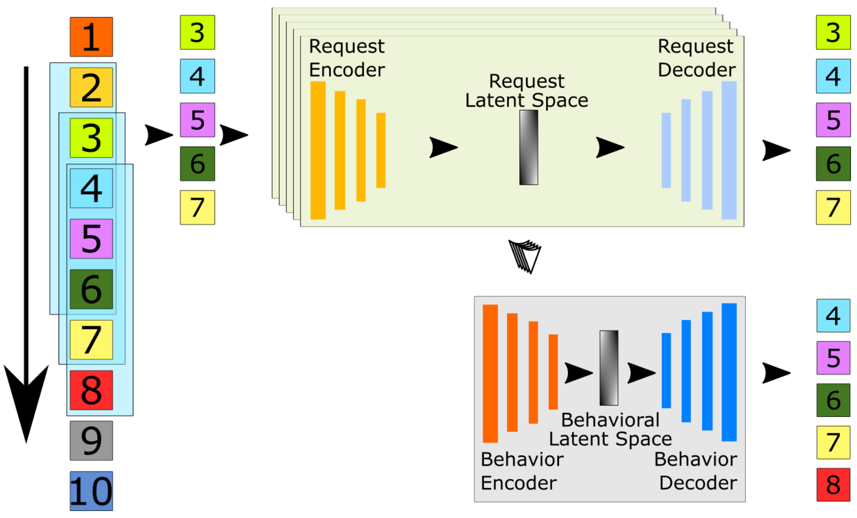

Figure 3.2.: Conceptual big picture of the developed architecture. The request encoder first reduces each request to a latent representation. The behavior encoder builds upon that compressed representation. Finally, the behavior decoder tries to predict the next sample. The request decoder is used during training to speed up the training.

we use separate instances for each request. The individual instances share their weights and biases of the layers. Sharing the parameters also keeps the model size small, which reduces the overall training time needed.

The sequence of compressed requests serves as input for the behavior Autoencoder. The behavior decoder does not reconstruct the exact input sequence. Instead, it reconstructs a sequence shifted by one request to the future. Shifting the output sequence implies that the Autoencoder will truncate the request farthest in the past. Additionally, the newest request is not part of the input sequence. The lack of presence forces the network to focus on behavior patterns in order to predict the new request successfully.

During the development of the architecture, we also tested an alternative behavior decoder. The alternative version reconstructed the next expected request only, which Figure 3.2 illustrates as the rectangle labeled with number eight. However, reconstructing the full sequence showed an improved detection performance.

Using the described two-step approach seems intuitive for humans, but it is hard to learn for a neural network without further modifications. Therefore, we added a request decoder that reconstructs the initial input sequence. Similarly to the request encoder, the decoder also uses a shared layer approach to ensure location independence within the sequence. The decoder improves the quality of the latent space representation early in the training process. Having good compressed representations early on makes the first training epochs of the behavior Autoencoder more valuable. Eventually, this improves the overall quality of the training result. The request decoder contributes to the training only and is of no purpose in prediction mode. Thus, removing the request decoder layers speeds up the prediction performance of our architecture.

3.2.2. General Architectural Details

Our detection system expects the input features grouped into two categories: *text input* and *meta input*. Text input consists of the *path*, *query*, and *referer*, so all features that build on text-based input. It has a total length of 620 tokens per request. Meta input, in turn, uses the categorical features *HTTP verb*, *day-of-week*, *time-of-day*, *request-time-delta*, *browser*, and *operating system*. The features sum up to a total length of 30 input nodes. Splitting the features into the two categories takes place in the input pipeline. Despite its name, the pipeline does not only provide the input data for the network. It also provides the target output data. The decoders use the same feature categories as the encoders. To sum up, the model expects two inputs (text input, meta input) and provides four outputs. We refer to the outputs as *request text output*, *request meta output*, *behavior text output*, and *behavior meta output*.

Dividing the features into text-based features and meta features is due to the different nature of the features. Meta features are categorical features represented in a one-hot encoding. The individual features do not contain any sequential information. Text-based features, in contrast, are a sequence of characters where the order does matter. Handling the features differently, therefore, potentially increases the Autoencoder's overall performance.

The output format of text features differs from its input format. The input layer accepts the tokens as integers, while the output layer generates one-hot encoded tokens. For meta features, both the input and output are one-hot encoded. The encoding of text features is necessary because tokens with close numbers are not necessarily related to each other, which would be the case when using embeddings. For example, if the target token has the value 222, it is not possible to tell in general if a prediction of 221 or 42 is closer to the target. However, changing a prediction from 222 to 42 would require bigger weight changes compared to a transition from 222 to 221. The one-hot encoding ensures that the transitions are equally expensive. We avoided the one-hot encoding of the input layer because this would result in a significantly larger network.

While developing the architecture, we found out that bidirectional GRUs are superior to unidirectional GRUs concerning reconstruction quality. A bidirectional setup processes the sequence twice. One GRU processes the sequence in a forward pass, the second GRU processes it in reversed order. *Addition*, *multiplication*, *average*, and *concatenation* are the standard techniques available to combine the results of the two passes. In our final architecture, all bidirectional GRUs concatenate the outcome of the forward and backward pass.

Throughout the following architecture description sections, we adopt the layer names used by Keras. We use Keras to implement the proposed architecture. While the majority of names are self-explanatory, we note that *Dense* layers describe a fully-connected layer. The *AttentionWithContext* layer used in the behavior Autoencoder is a custom layer not deployed with Keras. It implements the attention mechanism Yang et al. [Yan+16b] defined. For implementation details, we refer to Listing B.1 in Appendix B.

3.2.3. Request Autoencoder Details

Finding a good compression for the individual requests is the assigned task of the request Autoencoder. Figure 3.3 shows the Autoencoder we designed to accomplish this task. For the sake of illustration, we attach the transformation layers to the corresponding main layers. Listing A.1 and

Listing A.2 in Appendix A give detailed layer descriptions as generated by Keras.

The Autoencoder expects a sequence of requests as input. First, it splits the sequence into its individual requests. Then, it applies the encoder and decoder to each of the requests. Finally, the reconstructed requests form a sequence again. Ideally, the output sequence matches the input sequence exactly.

The request Autoencoder can only take information about the current request and none of the neighboring ones into account. At the same time, the request compression must work independently of the location. Location independence means compressing a request equally on any given sequence position. We prevent the mixing of information from individual requests by using a separate Autoencoder instance for each item of the sequence. The instances share their weights and biases to keep the location independence property intact. Additionally, sharing the parameters keeps the size of the model small.

The encoder first uses a Dense layer to reduce the sequence length for the following bidirectional GRU layer. The number of nodes the Dense layer uses is half the size of the input layer. Reducing the sequence length speeds up the training. Another Dense layer then combines the two input branches.

As far as possible and useful, the decoder is symmetric to the encoder. Therefore, the decoder also uses Dense and bidirectional GRU layers. We highlight two interesting aspects of the decoder. First, the last Dense layer of the text branch has the sole task of transforming the tokens of the generated sequence into a one-hot encoded representation. Second, the first Dense layer of the meta branch is overcomplete. It has 200 nodes, whereas the final output layer requires only 30 nodes. However, the Dropout layer following that large Dense layer has an untypical dropout rate of 0.2. SAEs inspired this structure, which allows individual nodes to focus on different features. The network can learn a large number of different data relations this way.

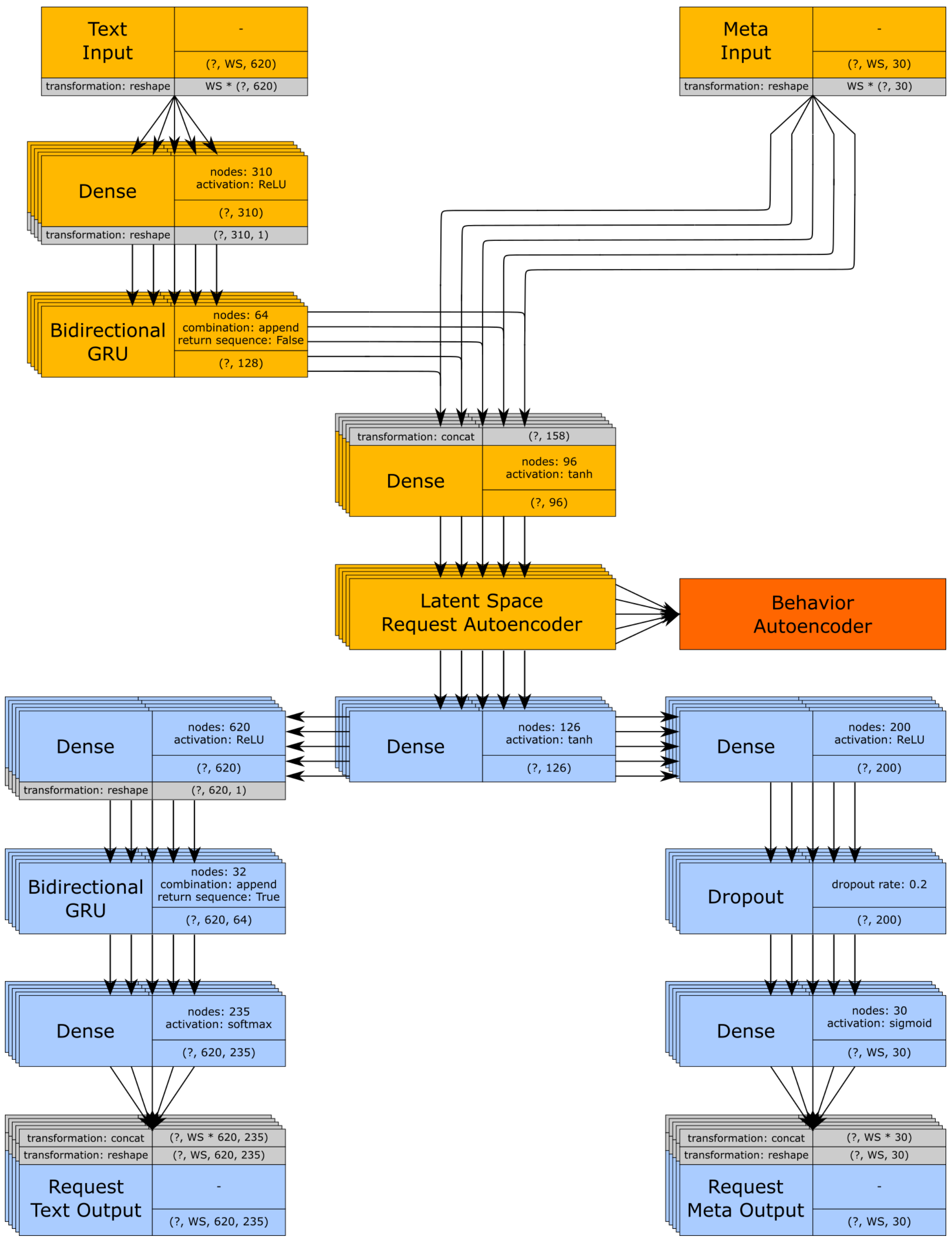

Figure 3.3.: Architecture of the request Autoencoder. WS stands for the selected window size.

3.2.4. Behavior Autoencoder Details

The task of the behavior Autoencoder is mining user behavior based on a sequence of compressed requests. We require the behavior Autoencoder to reconstruct the input sequence shifted by one request. This implies that the newest request to reconstruct is not part of the input. The shift prevents any memorization effects for input data and forces the Autoencoder to learn behavior. Figure 3.4 shows the layers used for the behavior Autoencoder. Listing A.3 and Listing A.4 in Appendix A give the layer description generated by Keras.

The behavior encoder expects the sequence of compressed requests as input. Therefore, a transformation layer combines the latent spaces of the individual request Autoencoders to a sequence. The bidirectional GRU returns not only the state after the last sequence item but all interim steps as well. An AttentionWithContext layer uses the interim steps and reduces them to a 128-dimensional latent space representation. AttentionWithContext is a custom layer not shipped with Keras. It implements the attention mechanism Yang et al. [Yan+16b] proposed. With this mechanism, the network learns on which areas of the sequence it must focus. The attention mechanism works context-aware. We refer to Listing B.1 in Appendix B for implementation details.

The decoder then reconstructs the full requests from that one-dimensional latent space. The branch of the decoder creating the text-based features and the branch creating the meta-features are not sharing any layers. They work independently of each other. The branch responsible for the text output first uses a GRU layer to create a sequence of requests. Next, a Dense layer transforms each of these requests, which are in no meaningful representation at this moment, into the correct one-hot encoded representation. The branch creating the meta-features consists of two Dense layers and a Dropout layer in between them. This Dropout layer's sole purpose is to prevent overfitting.

The performance of a model also requires a set of hyperparameters aligned with the architecture. The next section explains the model's hyperparameters. It gives details about the inspected ranges, as well as the finally chosen value for each of the discussed parameters.

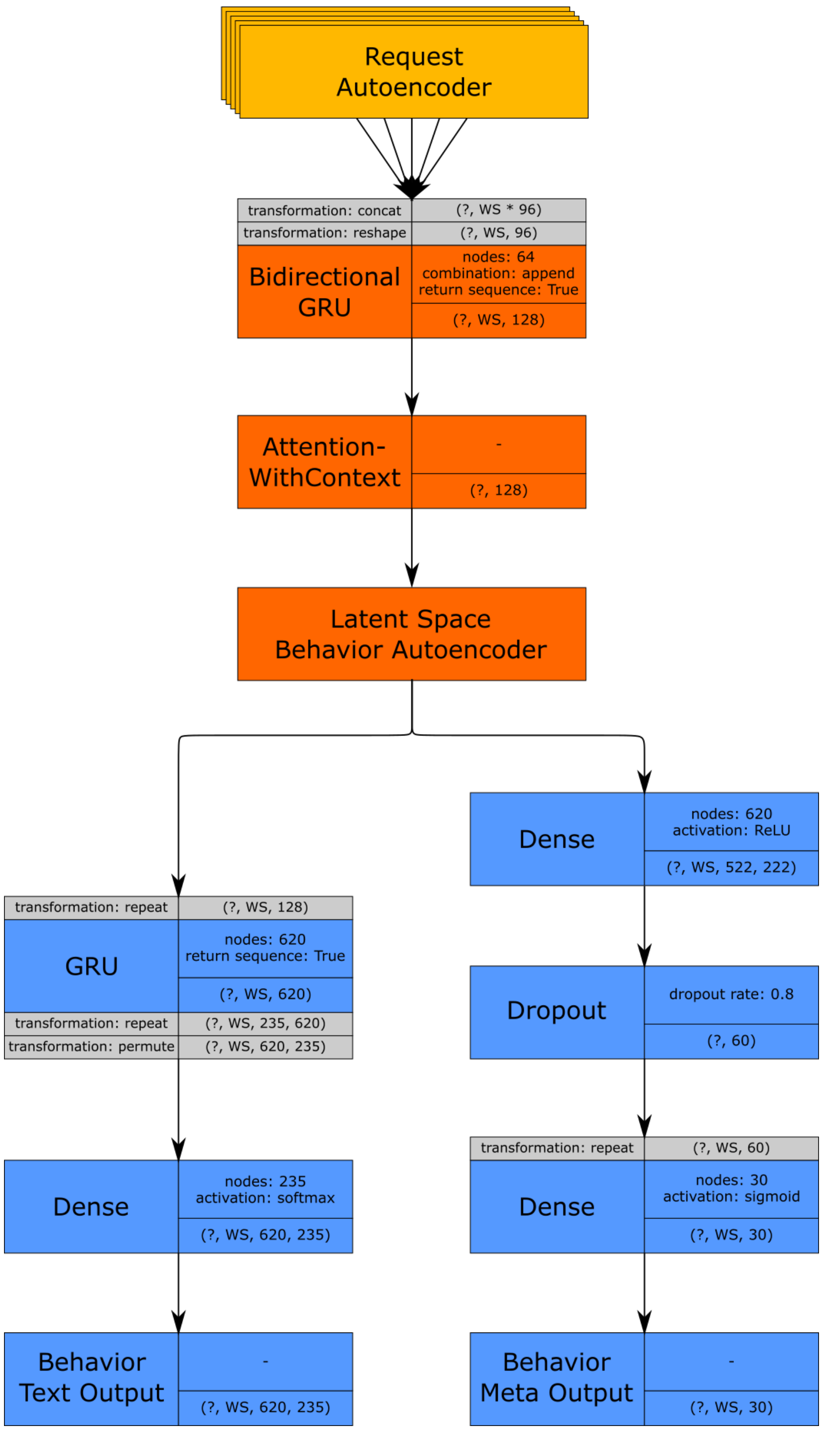

Figure 3.4.: Architecture of the behavior Autoencoder. WS stands for the selected window size.

3.3. Hyperparameter Search

Hyperparameters are crucial for a well-performing neural network. Finding a good set is challenging, as the number of hyperparameters explodes when increasing the complexity of the architecture. By using a random search-based approach, the chances of finding a well-performing set of hyperparameters increase. Random search does not guarantee to find the best set of parameters, but it usually finds a good one early in the process. The following paragraphs describe the tested hyperparameters and list the corresponding value ranges we inspected.

Batch size. The batch size strongly influences the time needed for training. We tested batch sizes of 32, 64, 128, and 256 samples. The batches are multiples of two, so the Graphical Processing Unit (GPU) can process them optimally. In terms of epochs, small batch sizes boost the decrease of the loss compared to large batch sizes. However, the total number of calculations per epoch increases with small batch sizes. More operations then increase the needed training time per epoch. A batch size of 64 delivered the best trade-off between the total number of operations and loss optimization speed.

Window size. The window size affects the model's ability to learn behavior. The longer the sequence, the more information about the usage behavior the model can extract. However, increasing the window size also increases the computational costs of the training. Window sizes tested during the architecture development process ranged from five to 20 requests. Using large window sizes caused memory exceptions for some of the more complex architecture candidates. Therefore, we use the final window size of five.

Activation function. Fully connected layers require a non-linear activation function. Figure 3.3 and Figure 3.4 show the activation functions for the individual layers. The output layer of the text-based features uses the softmax function. We chose the softmax because it favors single activations along a given axis, which fits the requirement of one-hot encoded data perfectly. The meta output layer represents a concatenation of several class-based features. Each class requires the activation of one node. Therefore, we use the logistic function. Its output is in the range $x \in [0, 1]$, which fits the requirements better than other activation functions. For interim layers, a combination of *tanh* and ReLU showed the best performance.

Loss function. A combination of loss functions quantifies the model's performance. For the two text output branches, we use the categorical cross-entropy. The loss of the two meta feature branches is calculated with the binary cross-entropy.

Optimizer. We use the optimizer Adam [KB14] for the parameter updates. Although it is computationally more expensive than SGD, it usually outperforms SGD and the other optimizers. As the initial learning rate, we use the one the paper suggests, which is 0.001.

4. The Dataset

The quality of the dataset used for any machine learning-based project influences the prediction performance significantly. Therefore, this chapter describes the origin of the data and the preprocessing steps performed. The chapter starts with the general aspects of datasets. They are different regarding scope, availability, and level of detail. It then describes the characteristics of the selected dataset. Additionally, it gives some general statistics about the set. Finally, the chapter explains the preprocessing steps required for each feature.

4.1. Dataset Selection Process

Selecting a dataset is one of the earliest tasks in any data-driven project. Notwithstanding, it is a very crucial task. The selected dataset significantly impacts the chosen preprocessing steps, algorithms, and corresponding hyperparameters. Therefore, we explain the key aspects of datasets in the following. We start by discussing general aspects. Then, we shift the focus towards log file-based datasets and their characteristics.

4.1.1. General Categorization of Datasets

Target knowledge, origin, and availability are three important properties of datasets. The properties apply to datasets in general and are not specific to our domain. The following section discusses the properties and essential aspects.

Labeled or Unlabeled. Labeled datasets contain a target output value for each sample. Assigning labels to the samples is work-intensive and sometimes even infeasible. Therefore, they are usually harder to obtain. Whether a dataset is labeled or unlabeled influences the set of algorithms available to choose from substantially.
Depending on the dataset, the correct label for a sample might not be evident. Thus, labeling is an error-prone task. Showing the sample to multiple persons and performing a majority voting reduces the number of wrongly labeled samples. Systems like *reCAPTCHA* [Von+08] force users to label new samples while still offering a benefit to them.

Real or Simulated. Generating a dataset that represents real data is challenging, independent of the specific domain. Additionally, simulated data might not represent real-world data. Generating the data is so complex that it is sometimes published separately from the work building on the data. The uncertainty about applying the results to real-world data encourages authors to commit to the dataset details early on. This way, they can rule out any dataset changes for a better performance of their model.

Public or Private. Sets are either available publicly or kept private. While most generated datasets are released to the public, real-world datasets are typically kept private. Typical reasons to keep the datasets private are privacy concerns and the fear of exposing sensitive data. However, properly anonymizing a dataset is challenging. Every change in the raw data could impact the performance of the model on non-anonymized data. Tuor et al. [Tuo+17] also argue that anonymizing data can alter relevant relations in the dataset. When the anonymization strategy contains flaws, an attacker can undo the process. For instance, Narayanan and Shmatikov [NS08] deanonymized users in a dataset of movie ratings released by the online movie streaming service Netflix[1].

[1]https://netflix.com

4.1.2. Characteristics of Cybersecurity Datasets

Numerous studies [YA17; SLG18; Rin+19] have evaluated the strengths and weaknesses of publicly available datasets. The authors focused on datasets intended for intrusion detection and cybersecurity-related tasks. To summarize their findings, one can claim that the quality, as well as the structure of the individual datasets, varies widely. This section highlights the most noteworthy differences between the datasets.

Dataset format. Different formats to store datasets are available. The best-suited format depends on the structure of the data. A frequently used format for network captures is the so-called *PCAP*[2] format. Most traffic recording tools available for Windows or Linux support the binary file format. In general, dumps of the raw network packets prefer the binary format. The structure of the packets is strictly defined, and numerous tools for reading and writing the packets exist. In contrast, application data like log files and processed data network flow summaries usually use a text-based format. Their structure varies widely between applications. Text-based formats are more convenient to work with for humans.

Scope. The scope of a dataset defines which data are included. Some datasets contain all network devices of an organization, others focus on subsets of clients, and others only contain information of a single client. Additionally, datasets can be limited to a single application. Often the datasets streamline the requests of multiple clients and bots into a single dataset. This work requires several requests per user to be present in the set. Otherwise, it is not possible to extract user behavior.

Level of detail. Datasets expose a significant variation in their level of detail. While some network-based datasets contain the full network traffic, others contain header information only. Yet other datasets alter the header information, too, during anonymization. In contrast, application log data contain no network header information whatsoever. Instead, they contain the application data missed in most network traffic-based sets. Therefore, which dataset suits best depends on the attack to be detected.

[2]https://www.tcpdump.org/pcap/pcap.html

Based on the presented characteristics, we selected a dataset for this thesis. The following section presents the dataset and provides some general statistics about the set.

4.2. Selected Dataset

Zaker [Zak19] recorded the dataset we selected for our work. We require a dataset containing valid traces of multiple users. It should not contain parameter brute-force attacks or comparable attacks. The dataset is publicly available and composed of log entries of an online store. The recording began on 22/Jan/2019 at 00:00 UTC and ended on 26/Jan/2019 at 17:00 UTC. The set was captured on an Nginx[3] web server and resulted in 10,365,152 records in total. Each line of the dataset file represents an HTTP request logged by the server in the *Combined Log Format*. The fields actually filled with values are *source IP*, *timestamp*, *HTTP verb*, *path*, *HTTP version*, *response status code*, *response size*, *referer*, and *user agent*. Table 4.1 shows one record present in the dataset as an example. The dataset contains 36,044 different source IPs that issued 40 requests on average. The source with the most entries counts 353,483 requests. The requests use the HTTP methods *GET* and *POST* only. Since the store was hosted on the Iranian top-level domain *.ir*, the log file contains Arabic characters in an HTML-encoded format.

4.3. Data Preprocessing

Preprocessing describes the process of transforming the raw base data into a format that suits a neural network. We start by splitting the log lines into individual features. The individual features undergo additional preprocessing steps based on their metric. During the next step, we group the requests: Requests from a single user within a defined period form a session. Finally, the sessions are split into a training, validation, and testing set. The following sections explain the process applied to the raw text-based

[3] https://www.nginx.com/

5.160.157.20 - - [22/Jan/2019:03:56:49 +0330] ”GET /browse/blu-ray HTTP/1.1” 301 178 ”-” ”Mozilla/5.0 (Windows NT 5.1; rv:8.0) Gecko/20100101 Firefox/8.0” ”-”	
Source IP	5.160.157.20
Date	22/Jan/2019:03:56:49 +0330
HTTP Method	GET
Path	/browse/blu-ray
Protocol Version	HTTP/1.1
HTTP Response Status Code	301
Response Size	178
HTTP Referer	-
User Agent	Mozilla/5.0 (Windows NT 5.1; rv:8.0) Gecko/20100101 Firefox/8.0

Table 4.1.: Example entry from the log file used as dataset [Zak19] for this work. It lists essential fields and provides the value of one entry as an example.

log lines. They also cover the preprocessing steps required for the individual features.

4.3.1. User Session Extraction

We start with an ordered log file, containing all requests to extract user sessions. The first step splits this stream of requests into user sessions. We identify users based on their IP address. A filter removes sessions that origin from bots. Another filter removes requests that do not contain a substantial number of requests. Short sequences do not expose learnable usage behavior. Request parsing and filtering reduce the total number of requests to 1,869,536. The following paragraphs explain the individual steps in more detail.

Line Parsing

The log file contains one request per line. We rely on regular expressions to extract the individual fields from each request line. Table 4.2 shows

Source IP / Hostname	`[0-9a-zA-Z.\-_]+`
Date	`\[([0-9:/A-Za-z]+ (-|\+)[0-9]+)\]`
HTTP Method	`(GET|POST|PUT|DELETE|PATCH)`
Path	`([0-9a-zA-Z/_\-.%|,]+)`
Query String	`(\??[0-9a-zA-Z/_\-.=?&^*%@\\\|,+:() \[\]{}'! ;~$§"<>#]+)?`
Protocol Version	`HTTP/([0-9\.]+)`
HTTP Status Code	`([0-9]{3})`
Response Size	`([0-9]+|-)`
HTTP Referer	`"([^"]+)"`
User Agent	`"([^"]+)"`

Table 4.2.: Regular expressions used for the individual fields during feature extraction.

the regular expressions. They do not match 36,044 records. Non-matching records either use an unsupported HTTP method or are not valid HTTP requests at all. Supported HTTP methods are *GET*, *POST*, *PUT*, *PATCH*, and *DELETE*. *HEAD* and *OPTIONS* requests are present in the dataset but not matched by the parser because they represent preliminary requests. Requests with one of the supported verbs follow such requests. The fields extracted from each log line are the *HTTP method*, *path*, *query-string*, *user agent*, *timestamp*, *referer*, and *time of the request*.

Session Extraction

In the next step, we extract sessions from the streamline of requests. IP addresses serve as an identifier to distinguish between different users and devices. Unfortunately, IP addresses alone do not provide sufficient information to identify users uniquely. For example, networking techniques such as Network Address Translation (NAT) hide numerous users behind a single public IP address. However, IP addresses are the best-suited criteria present in the dataset. In the selected dataset, it is not possible to track users across different IP addresses due to the lack of a user identifier token. Users may end up using different IP addresses when they switch from their mobile data connection to Wi-Fi, for example.

A period of 30 minutes without any request from the user closes a session. Requests received after an idle time open a new session. Compared to the approach presented by Jiang et al. [Jia+16], splitting the sessions of the same source based on the idle time only is a somewhat naïve approach. Jiang et al. use additional information like an empty referer to detect the starting point of new sessions. However, the dataset does not contain a ground truth for user sessions. This way, it is not possible to tell whether a more sophisticated approach delivers better-divided sessions at all. We decided to stick to the basic approach for this reason. The user session splitting leads to 337,874 identified sessions with an average length of 31 requests per session.

Session Filtering

A session has to contain a minimum number of requests so our system can learn usage behavior characteristics. Therefore, a filter excludes all sessions containing less than 20 individual requests. Another filter drops sessions exceeding a total of 100,000 requests. The filter found two sources where the number of requests is that large. Additionally, the filter drops sessions with no idle time for several hours. We consider this as an indicator for multiple users sharing an IP address. As this distorts the user behavior, we decided to exclude these users. A third filter removes all sessions caused by bots or web crawlers. The filter relies on the user agent field, as crawlers usually identify themselves with a custom user agent. In contrast to humans, crawlers aim to visit all subpages of a homepage. Their browsing patterns differ from the patterns humans expose. Attackers could try to bypass the protection system by faking a crawler's user agent. Because critical data require prior authentication, crawlers should not be able to access such data. Thus, an attacker would learn only about information already available publicly. It is crucial to filter the requests before splitting the data into training, validation, and testing sets in order to keep the size ratio of the sets intact. Table 4.5 lists the number of requests for each identified bot. The filtering left 1,869,536 requests separated into 50,690 sessions.

4.3.2. Set Splitting

After extracting the user sessions, we partition the sessions into a training, validation, and testing set. The first fifth of the whole dataset represents the testing set. The remaining larger part serves as the source for the other two sets. One fifth represents the validation set, and the other four fifths represent the training set. We apply the ratio to the sum of requests, and not to the number of sources. As a result, the training set ends up being the largest set with 64.95% of data from the filtered dataset. It contains 32,938 sources and 1,214,280 requests. The validation set contains 15.87% of the filtered dataset. It consists of 8,045 sources, with a total of 296,434 requests. The testing set contains 9,707 sources and 358,822 requests.

4.3.3. Feature Preprocessing

An important subtask of preprocessing is transforming the features into the target format. One field present in the log file can serve as a source for more than one feature used as input to the neural network. For example, we extract features like the *weekday*, *day-of-year*, or *week-of-year* from the request timestamp.

Common other tasks include transforming text into a numeric representation, as well as standardizing or normalizing data. When standardizing data, they are transformed to expose a standard deviation σ of 1 and a mean μ of 0. In contrast, when normalizing data, the value range is changed to be in the interval $[0;1]$. Possible value ranges are only selected from the training and validation set. The final model evaluation is the only phase that can use the test set. Any selected hyperparameters rely on the other two sets only.

In the following, we describe the extracted features. We also explain the applied preprocessing techniques. First, we handle all features transformed into a categorical representation. Afterward, we give details about the text-based features.

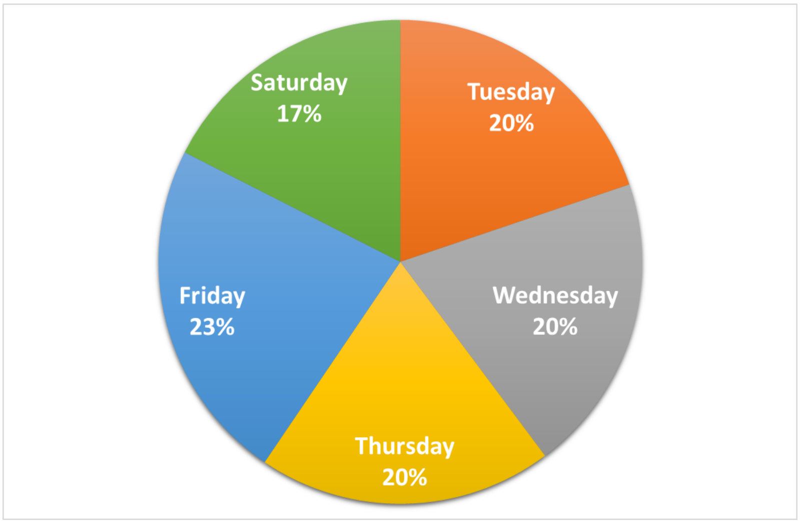

Figure 4.1.: Distribution of requests over weekdays. The dataset recording started on Tuesday and lasted for four more days. For this reason, the chart does not contain all weekdays.

Categorical Features

The first extracted feature is the HTTP request method. Initially, the data extraction logic supported *GET*, *POST*, *PUT*, *PATCH*, and *DELETE* verbs. However, as the dataset only uses *GET* and *POST*, we considered only those two in the subsequent steps. This decision keeps the network size smaller.

The timestamps of the HTTP requests serve as the data source for three different features. First, we put requests into classes based on the day of the week they were sent. While the behavior of users is likely the same during working days, it might be different on weekends or holidays. Figure 4.1 shows the distribution of requests over weekdays. As the request collection period lasted for five days only, the dataset does not contain requests for every day of the week.

Another extracted feature is the request timestamp. It describes the time a user issued a request. We decided to use four classes. Each class contains requests of six hours. The borders that separate the classes are at 06:00 AM, 12:00 AM, 06:00 PM, and 12:00 PM.

Delta Range	Test	Valid	Train	Total
$0 \leq \Delta t < 10$	339,997	279,139	1,145,372	1,764,508
$10 \leq \Delta t < 120$	16,237	15,003	59,241	592,417
$120 \leq \Delta t < 660$	2,019	1,759	7,253	42,271
$660 \leq \Delta t < 960$	249	224	1,008	1,481
$960 \leq \Delta t \leq 1800$	320	309	1,405	2,034

Table 4.3.: Classes of idle time between two subsequent requests of a user, including the number of assigned requests. The maximum value possible is 1,800 seconds (30 minutes). Idle times exceeding that limit trigger the creation of a new session.

Class Name	Test	Valid	Train	Total
Unknown / Other	8,605	7,994	31,555	48,154
Firefox	35,172	28,279	122,726	186,177
Chrome	260,594	215,322	875,728	1,351,644
Edge	-	-	-	-
Internet Explorer	805	788	3,652	5,245
Safari	53,646	44,051	180,619	278,316

Table 4.4.: Number of requests issued by the given browser on a per-set basis. Browsers identified based on the user agent.

Finally, we use the time difference between the current and previous request as a feature. Although the time difference could be represented as a number, too, we decided to use classes of deltas. The classes reduce the effects of timing differences caused by the network or devices between client and server. Table 4.3 shows the defined classes and gives the number of samples per class. It also shows the idle time ranges used to classify the requests.

The last two features rely on the user agent field. The first describes the device of the user. The second the used browser. Detecting the users' browser and device relies on a straightforward approach. Our system checks the presence of terms associated with a given browser or operating system. It extracts browser and device information separately. Table 4.4 lists the classes available for browsers and also shows how many entries belong to a given class per set. Similarly, Table 4.5 shows the classes per device. Table 4.5 includes bots as well that we filtered earlier in the preprocessing pipeline.

Class Name	Test	Valid	Train	Total
Unknown / Other	3,078	4,175	18,011	25,264
Windows NT	65,246	51,252	214,519	331,017
Macintosh	793	696	3,057	4,546
Linux	736	755	2,856	4,347
Android	251,420	209,088	850,188	1,310,696
iPhone, iPad	37,549	30,468	125,649	193,666
Googlebot	-	-	-	801,797
Bingbot	-	-	-	197,769
AhrefsBot	-	-	-	57,178

Table 4.5.: Number of requests originating from the given device or system. System detection is based on the user agent. No numbers per set are available for bots and crawlers because these requests were excluded before the splitting took place.

Text-Based Features

In general, neural networks need numeric input. However, path, query, and referer features are text-based. Hence, they need to be transformed first. A standard approach to transform text-based features into a neural network-friendly format is to tokenize the text. Embeddings can enhance the representation of these tokens. Tokenization supports word-based and character-based approaches. For word-based tokenization, advanced techniques can detect names of institutions, places, and objects. However, the simple variant is sufficient for URL tokenization.

We refer to the set of words known by the tokenizer as *vocabulary*. To prevent it from becoming exorbitantly large, we introduce thresholds. They ensure that the vocabulary consists only of words that occur more than a specified number of times. Limiting the vocabulary size is more critical for word-based tokenization. Nevertheless, Table 4.6 shows vocabulary sizes for different thresholds for both approaches. We order the tokens by occurrence so that less frequent ones can be removed later in the process if needed. Tokenizers should access only the training set and validation set to build the vocabulary. The lowest indices typically represent special tokens like *UNKNOWN* or *PADDED*. We performed both word-based and character-based tokenization to keep flexibility during architecture development.

	≥ 1	≥ 2	≥ 3	≥ 4
Character-based	234	232	215	211
Word-based	310,654	71,682	58,750	53,054

Table 4.6.: Vocabulary sizes for different minimum numbers of occurrence of tokens. Shows numbers for character-based and word-based tokenization.

	Total Lengths			**Chosen**	**Percent not Truncated**		
	Test	**Train**	**Valid**	**Length**	**Test**	**Train**	**Valid**
Path (char)	163	397	160	140	99.99%	99.99%	99.99%
Query (char)	5,401	5,394	5,206	160	98.62%	97.12%	99.16%
Referer (char)	1,229	1,414	1,329	320	99.99%	99.99%	99.99%
Path (word)	31	78	33	32	100.00%	99.99%	99.99%
Query (word)	561	569	559	128	99.88%	99.87%	99.92%
Referer (word)	146	230	118	96	99.99%	99.99%	99.99%

Table 4.7.: Comparison of tokenized sequence lengths for the features *path*, *query*, and *referer*. It shows the maximum length, our chosen length, and the percentage of requests entirely representable with the chosen length per set.

We determined the maximum length of the path, query, and referer features to account for varying lengths of URL sequences. This is needed for designing a network capable of handling the request sequences. Sequences tokenized with a character-based approach tend to be longer than sequences tokenized in a word-based manner. Table 4.7 shows the maximum value found in the corresponding set. The table also shows the length we chose for our architecture and for which percentage of requests this is still sufficiently long for full representation.

In the next step, we evaluate the performance of our proposed system. We use the dataset discussed in detail in this chapter. The model first goes through a training phase. Following the training phase, we evaluate the detection performance. The next chapter explains in detail the steps performed, as well as the metrics used.

5. Evaluation

In this chapter, we focus on the outlier detection performance of the developed system. The chapter first explains how we trained the model and which technologies we used. In addition, we list the hardware details of the system in charge of the training. We then discuss the methodology used for assessing the performance. The evaluation is based on two capabilities. First, we require a low reconstruction error for valid sequences. Second, the model must detect anomalous behavior. Following that, we introduce the metrics required for assessing the model's performance. The final section shows plots of the model's performance and interprets them. We also discuss some architecture candidates we tested that did not perform as expected.

5.1. Neural Network Training

Improving the performance of a neural network needs many epochs. The network processes the full training set once per epoch. Furthermore, it improves its generalization capabilities during each run. The sum of all epochs is called the training phase. The following section first focuses on the tools used for implementing the training framework. It then describes the hardware setup. Finally, it shows how the reconstruction loss and the reconstruction accuracy developed over the epochs.

5.1.1. Development Tools

We developed a framework based on TensorFlow and Keras. The framework loads a model definition, performs the training, and logs valuable information. Valuable information includes (1) training progress concerning loss and accuracy, (2) epoch durations, (3) run configuration details for reproducibility, (4) model architecture details as an image as well as in the machine-readable *JSON* format, and (5) checkpoints of the model created during training. It further supports exchangeable loss functions and early stopping. The framework accepts parameters via the Command Line Interface (CLI) to change the default values.

TensorFlow[1] serves as a foundation for creating and training the neural network. It is developed by Google and is a dataflow graph-based library for numerical computations. TensorFlow is popular for machine learning-related tasks. Although funded and developed by Google, its code is available publicly. For many operations, TensorFlow includes kernels targeting Computer Processing Units (CPUs), Graphical Processing Units (GPUs), or Tensor Processing Units (TPUs). During execution, the library dynamically chooses the correct kernel depending on which device it schedules the operation. TensorFlow also provides an input pipeline in the *tf.data* namespace. It represents a fast way of loading the data and transforming them into a format accepted by the model.

TensorFlow supports distributed training on multiple servers. As models independent of the problem domain tend to increase in size, distributed training became a key advantage of TensorFlow. Therefore, updates of the library optimize its distribution strategy. Raschka and Mirjalili [RM19] list major changes between TensorFlow 1.X and TensorFlow 2. The most severe change targets the computation graph creation. The library determines the best order of execution for the operations based on the computation graph. It also tries to distribute the operations across the available devices ideally. With TensorFlow 2, the maintainers enhanced the auto-generation of the graph. This allows developers to define models in a way more native to python compared to the old version.

Besides TensorFlow, which provides access to the low-level operations, we also use Keras[2] for implementation. Keras is a neural network abstraction framework that provides common layer types ready to use. Custom layers integrate into the library seamlessly. Initially, Keras supported other

[1]https://www.tensorflow.org/
[2]https://keras.io/

backends beside TensorFlow, too. However, TensorFlow 2 fully integrates Keras and should be favored over the multi-backend version. Géron [Gér19] defines Keras as being a Deep Learning API that eases the development and execution of models. The literature refers to the TensorFlow specific version as *tf.keras*.

5.1.2. Hardware Setup

Training large amounts of data requires a reliable hardware setup for optimal performance. Initial tests for a new architecture are feasible with a notebook. However, training the full dataset is not. A single epoch can last for several hours, and training lasts for several epochs. Hence, training a model on a consumer device may entail interruptions of the training process. Server-side training is prevalent in avoiding the mentioned problem. Besides that, dedicated GPUs for datacenters exist that speed up the training.

For our work, we had access to two server clusters to train our neural network. First, the institute runs a cluster on which we trained the early architecture models. Later, we used servers running on the Google Cloud Platform (GCP). Table 5.1 shows the specifications of the hardware. We also list the *CUDA Compute Capability*. It tries to make GPUs easily comparable with a single metric. The higher the value, the more instructions the GPU supports. More instructions available usually leads to better training performance.

The GCP supports different graphics cards depending on the region [Goo20]. We chose the listed GPU because it provides the best price-performance ratio. Which card performs best strongly depends on the type of layers used by the model to be trained. Dettmers [Det19] defines the following order of precedence for convolutional and recurrent networks, respectively:

Convolutional networks and Transformers:
Tensor Cores > FLOPs > Memory Bandwidth > 16-bit capability

Recurrent networks:
Memory Bandwidth > 16-bit capability > Tensor Cores > FLOPs

	Notebook	IAIK Cluster	GCP
CPU Model	i7-7820HQ	i9-9900K	var.
(v)CPUs	4	16	6
Clock Speed *[GHz]*	2.9	3.6	2.0 - 2.6
RAM *[GB]*	32	64	44
Graphics Card	GeForce Quadro M2200	GeForce RTX 2070	Tesla T4
CUDA Capability	5.2	7.5	7.5
VRAM *[GB]*	4	8	16
VRAM Type	GDDR5	GDDR6	GDDR6
Clock Speed *[MHz]*	1036	1620	1590
Memory Bus *[bits]*	128	256	256
Bandwidth *[GB/s]*	88	448	320
FP16 *[TFLOPS]*	-	14.93	65.13
FP32 *[TFLOPS]*	2.122	7.465	8.141
FP64 *[GFLOPS]*	66.3	233.3	254.4
CUDA Cores	1024	2304	2560
Tensors	-	288	320

Table 5.1.: Hardware specifications of used training platforms. Google automatically chooses the CPU model for instances running on the Google Cloud Platform (GCP). It runs on the Intel Skylake CPU platform or one of its predecessors. GPU specifications compiled from TechPowerUp [Tec17; Tec18a; Tec18b].

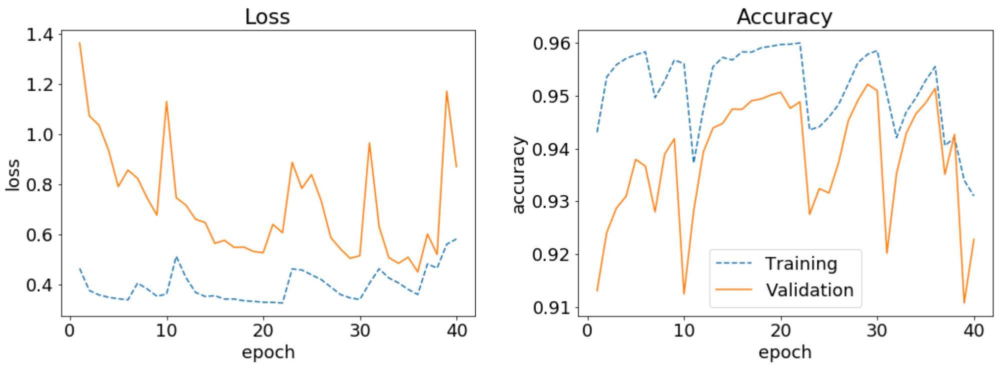

Figure 5.1.: Loss and accuracy development of the request Autoencoder.

Each individual building block of the architecture requires different hardware capabilities. Hence, cloud platforms are especially valuable in the initial architecture development phase. The testing of numerous different architectures is unavoidable in the initial phase. The option to change the underlying hardware easily supports the training process ideally.

5.1.3. Training Phase

We started the training process with a defined target of 20 epochs. This number of epochs was sufficient in preliminary runs on a subset of the data. However, the development of the loss indicated that the optimal capacity was still not reached after the 20 epochs. The model resided in the region of underfitting. Therefore, we resumed the training for another 20 epochs. When resuming the training, the random initialization values are replaced with the last model state. At the end of the resumed training, the model showed the first indices of overfitting. The generalization gap started to increase. Figure 5.1 shows the development of the loss and accuracy of the request Autoencoder. Similarly, Figure 5.2 plots the loss and accuracy of the behavior Autoencoder. In total, the full training with 40 epochs needed 6d 4h 32m on the GCP. Consequently, one epoch took approximately 3h 42m. The evaluation script ran for additional 21h 15m on the GCP.

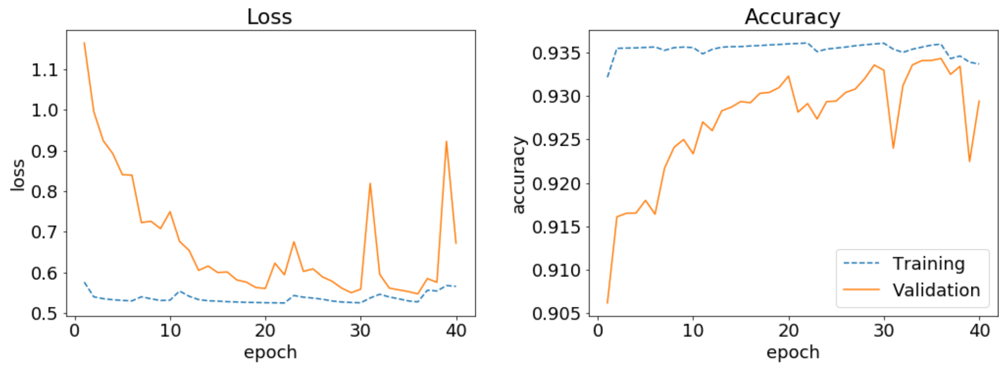

Figure 5.2.: Loss and accuracy development of the behavior Autoencoder.

Before starting the training, we configured a checkpoint callback provided by Keras. The callback creates a copy of the current state of the model after each epoch. Alternatively, it can create copies only if the loss improved. We decided to store copies after each epoch based on an observation: In some cases, the accuracy increases while the loss stays the same or is negligibly worse. However, the model with a slightly worse loss but significantly better accuracy likely performs better in detecting anomalies.

Based on the performance of the two Autoencoders, the models after 22 and 36 epochs were the final candidates. We finally selected the model after 36 epochs. We prioritized the performance of the behavior Autoencoder over the performance of the request Autoencoder. The selected model performs better on the validation set than the other model. This indicates that the model generalized better due to the additional epochs.

The metrics continuously improved with only a few minor exceptions for the first 20 epochs. This changes for the remaining 20 epochs. A likely reason for this behavior is the resuming of the training. The optimizer state was lost during this step. This means the learning rate for the individual parameters was reset to the initial value. Nevertheless, the trend of the loss and accuracy indicates that a new training run without interruption would not lead to better results. However, we note that the curves would have developed smoother if we had saved the optimizer state as well.

5.2. Evaluation Methodology

The following section explains the method used to evaluate the performance of the detection system. Since Autoencoders try to reconstruct the input, the reconstruction error is the key metric used for classifying outliers. However, the absolute value of the reconstruction error relates to the architecture of the model and is not informative on its own. Despite the architecture, the cost function also influences the absolute error. Percentages of anomalous classified samples fit the needs better than absolute error values when comparing different models.

The evaluation approach is split into two subareas. First, we show that the detection system is capable of learning behavior and applying the learned information to previously unseen data: the test set. Following that, we use a set of anomalous behavior traces and show that the average reconstruction error is larger compared to the test set. The larger reconstruction error indicates that the detection system can detect anomalous traces as such. We also explain how we generated the anomalous traces.

5.2.1. Generalization Capability

The first evaluation criterion shows the model's capability to generalize based on the training set and the test set. Therefore, we start with querying the model with a set of samples it has already seen during the training. We select a threshold based on the reconstruction errors measured at this step. The threshold defines the maximal reconstruction error allowed for a benign sequence of requests. Samples with higher reconstruction errors are classified as anomalous request traces. We select a threshold so that 99.9% of the request sequences seen during the training produce a lower reconstruction error. We chose such a high percentage because we expect the input dataset not to contain any invalid sequences. However, the network might struggle to reconstruct individual request traces for undefined reasons. Allowing a low number of such requests minimizes the overall impact of this issue.

Following that, we feed the test set into the model. It classifies the samples as malicious or benign based on the threshold determined earlier. A slightly higher number of anomalous samples is expected due to the generalization gap. However, the better the model generalized, the lower the difference between the training set and test set gets.

Although the presented evaluation approach shows the model's ability to reconstruct the input sequence, it does not guarantee a well-functioning anomaly detection system. If the model did not generalize well, it might simply reconstruct known samples as bad as unknown ones. Therefore, we show that the system can detect anomalies with the next set.

5.2.2. Outlier Detection Capability

The second criterion evaluates the model's ability to detect outliers. However, the original dataset contains only valid sequences of requests. Therefore, we generated an anomalous set of user traces. We obtained a labeled dataset by combining the anomaly set and the training set. Based on the labeled dataset, we can determine how many samples our system classifies correctly.

Generating anomalous traces is challenging. We decided against generating anomalous traces from scratch. Instead, we take valid traces and change parameters. Anomaly detection rates based on a generated dataset must be taken with caution for two reasons. First, making only minor changes to the requests might keep them too similar to valid ones. Second, changing the requests too much might result in requests no longer representing real-world ones. We decided to include this metric to show the detection of anomalies, and also decided not to change requests too drastically.

The test set serves as ground truth for valid user sessions. After applying filters, the total number of sessions remaining was 9,707. Session candidates must have at least 20 requests and at most 60. Next, we randomly select one sequence of requests for each session. Henceforth, we refer to this selection of requests as a trace. The length of each trace equals the window size. We then apply the changes explained in the following scenarios. The scenarios result in a total of 94,340 traces. In contrast, the test set consists of 1,082,528 traces. Thus, the combined dataset contains 8,02% of anomalous requests.

Scenario 1: Change Browser and Device Class

The first scenario focuses on credential abuse attacks. An attacker abusing credentials is likely to use a different browser or operating system. This scenario enforces an additional constraint on the set of session candidates. The traces must use the same browser class and device class throughout the whole session. Enforcing this requirement reduces the set to 9,252 valid sessions. In total, this scenario contributes 55,512 anomalous traces to the outlier set. We apply four different substitutions to the trace candidates.

First, we replace the browser value with a randomly selected other browser. We do not incorporate any heuristics or restrictions to the browser selection. We substitute the browser value on all requests of the selected trace.

Second, we apply the same substitution just explained with the browser category to the device category. The restrictions described earlier also apply to the used devices.

Third, we combine the first two variants. Using a different browser when using a different device is realistic. One might use Chrome on the mobile device, and Firefox on the computer.

Finally, we again change the browser and the device category. However, in contrast to the first three variants, we alter only the last request of the trace this time.

Scenario 2: Move Requests to Weekends and Vice Versa

The second scenario builds on the hypothesis that users behave differently on weekends compared to weekdays. Users cannot use the online shop while at work, for example. Therefore, we change the date of requests issued on a day during the week to weekend days and vice versa. Since we change only the day of the requests, other information like the time difference between two requests stays intact. This scenario adds additional 9,707 traces to the set of generated requests.

Scenario 3: Change Time Delta Between Requests

In the third outlier scenario, we alter the time deltas between the requests of a trace. We use two different variants. First, we randomly select a different time delta class for each request. Second, we select a different delta for the first request and apply this class to the remaining requests. The scenario builds on the hypothesis that an attacker misusing someone's credentials exposes a different behavior when issuing requests. The attacker might not be familiar with the user interface, for example. Hence, the attacker has to search the correct menu items to click on. Alternatively, the attacker might have automated certain actions. This scenario adds 19,414 traces to the generated set.

Scenario 4: Change Request Trace

The final scenario alters the request trace. For each trace, we randomly remove one request. Removing one request from the observed behavior likely indicates deviating behavior. However, this is not guaranteed as the randomly selected request could represent an optional step the user does not always perform. This scenario adds 9,707 samples to the set of generated requests.

5.2.3. Common Performance Metrics

By combining the training set with the anomaly set, we obtain a labeled dataset. In the following, we describe the metrics typically used to assess a model's performance on a labeled dataset. The metrics typically originate from statistics. Vinayakumar et al. [Vin+19] give a comprehensive summary of the most common metrics. Different fields of research developed the metrics independently of each other. Hence, many metrics have more than one valid name.

The metrics use a confusion matrix of a binary classification problem as their data source. Our approach also produces a binary output: The system should classify samples of the test set as benign and samples of the generated set as

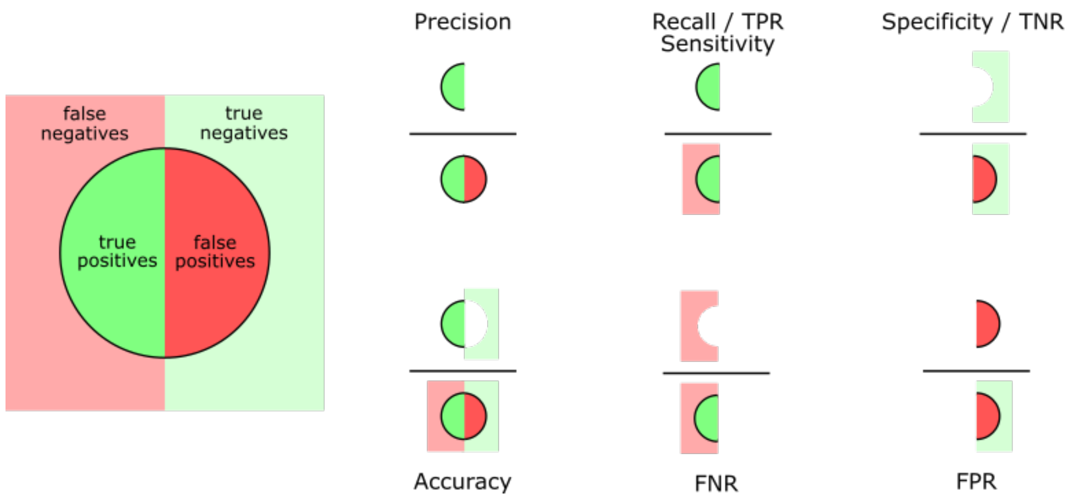

Figure 5.3.: Overview of different metrics and how to calculate them.

outliers. In a confusion matrix, one axis represents the actual labels, whereas the second axis represents the labels predicted by the system. Hence, four metrics build directly on that property:

- *True Positives (TPs)*: Samples correctly classified as outliers
- *False Positives (FPs)*: Samples erroneously classified as outliers. In statistical contexts known as *TYPE I error*.
- *True Negatives (TNs)*: Samples correctly classified as benign
- *False Negatives (FNs)*: Samples erroneously classified as benign. In statistical contexts known as *TYPE II error*.

The following metrics use the four classes of entries of the confusion matrix and express relations between the class sizes. The range of valid values for the metrics is defined as $x \in [0, 1]$. If not stated otherwise, higher values indicate a model performing better than models with lower values. For a better understanding of the relation of the metrics to each other, Figure 5.3 illustrates the most important ones.

Accuracy describes the ratio of correctly classified samples to the total number of samples. If one class is overrepresented, the accuracy is not an optimal metric choice. The accuracy of unbalanced classes is skewed, as a learning algorithm could decide to classify all samples as belonging to the larger class simply. In such cases, precision and recall give a more accurate picture of the model's performance.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

Precision or *Positive Predictive Value (PPV)* measures the portion of correctly identified outliers from the total number of samples classified as being outliers. The higher the precision, the more elements reported as outliers are actually outliers.

$$precision = \frac{TP}{TP + FP} = PPV \tag{5.2}$$

Recall, *Sensitivity*, or *True Positive Rate (TPR)* describes the ratio of correctly identified outliers to the total number of outliers present in the set. The higher the recall, the more anomalous samples the classifier is capable of detecting correctly.

$$recall = \frac{TP}{TP + FN} = TPR \tag{5.3}$$

Specificity or *True Negative Rate (TNR)* specifies the ratio of how many negative samples are actually classified as such.

$$TNR = \frac{TN}{TN + FP} \tag{5.4}$$

The *F1 score* measures the precision and recall of a classifier. A good performing classifier achieves good rates in both metrics. The F1 score combines the two metrics and is the harmonic mean.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.5}$$

The *False Positive Rate (FPR)* acts as a counterpart of the TNR. The better the classifier works, the lower this metric gets. It describes the proportion of incorrectly identified positives to all actually negative samples.

$$FPR = \frac{FP}{FP + TN} \tag{5.6}$$

The *False Negative Rate (FNR)* acts as a counterpart of the TPR. Again, a low value for this metric indicates a well-performing classifier. It describes the proportion of incorrectly identified negatives to all samples that are actually positive.

$$FNR = \frac{FN}{FN + TP} \tag{5.7}$$

The *Receiver Operating Characteristic (ROC)* curve is a plot of the TPR and FPR for different thresholds. The x-axis shows the values for the FPR; the y-axis shows the values for the TPR. Comparing models is possible by calculating the *Area Under Curve (AUC)* of the ROC curve.

5.3. Results

In this section, we discuss the performance of our outlier detection system. We use the two capabilities explained earlier in this chapter. We first show plots of reconstruction errors for different input sets. These plots confirm the system's capability of reconstructing behavioral patterns. Following this confirmation, we evaluate the anomaly detection capability. We rely on precision, recall, F1 score, and the ROC curve for this task. Based on the F1 score, we select the threshold used for the scenario evaluation. The detailed scenario evaluation shows that the model cannot detect all scenarios equally well.

5.3.1. Generalization Capability

First, we show that the model has generalized successfully. We feed all training data into the model and record the reconstruction errors. We then define a threshold above 99.9% of the samples. For our evaluation environment, we define a threshold of 0.0676. Figure 5.4 shows the reconstruction errors for samples of the training set. We use such a high percentage because the training dataset contains valid traces only. Accepting a few outliers at this step reduces the number of false positives. It is still possible that individuals performed some unexpected tasks.

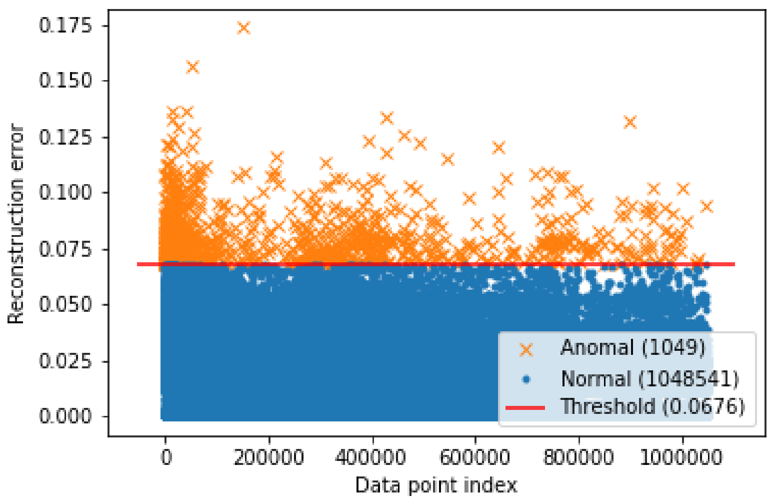

Figure 5.4.: Reconstruction error for training samples.

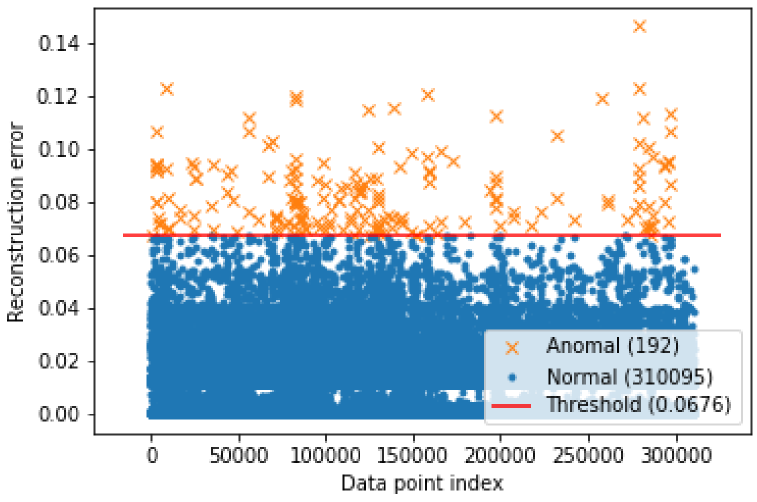

Figure 5.5.: Reconstruction error for previously unseen samples.

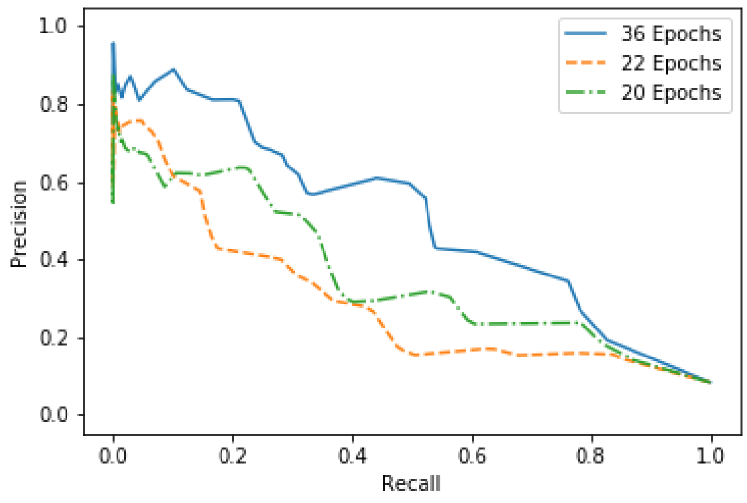

Figure 5.6.: Precision-recall curve.

Next, we query the model with the test set. The test set is unused up to this point. Thus, the model must predict behavior for previously unseen samples. Figure 5.5 shows the reconstruction errors. Based on the threshold defined earlier, the system classifies the traces as benign or outlying. Since only 0.06% of the samples are outliers, we conclude that the model has successfully generalized. It was able to learn user behavior. The model performed better on the unseen samples than on the training data.

5.3.2. Outlier Detection Capability

We feed valid and anomalous traces into our model to evaluate its outlier detection capabilities. The combination of the two sets forms a labeled dataset. All samples from the training set belong to the benign class. The samples from the generated set are anomalous. This way, we can use the metrics precision, recall, F1 score, and ROC curve. Based on the F1 score, we define the threshold for evaluating the individual scenarios. The F1 score ensures a good trade-off between precision and recall.

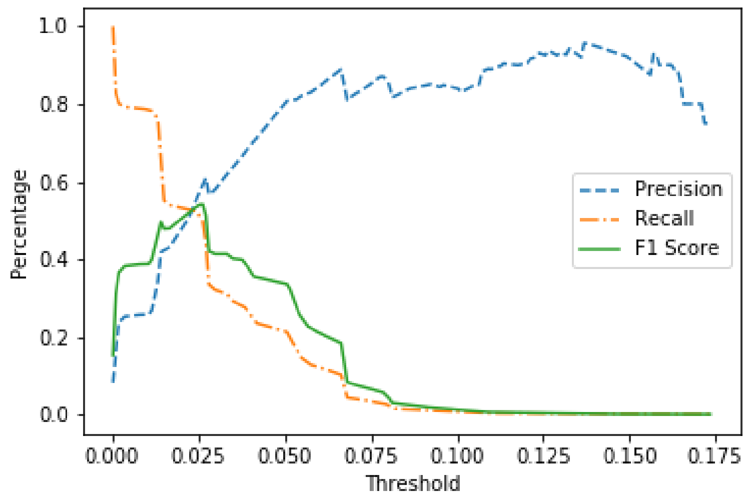

Figure 5.7.: Comparison of precision, recall, and F1 score.

First, Figure 5.6 shows the achieved precision and recall of the fully trained model after 36 epochs. Furthermore, it includes the models after 20 and 22 epochs. Improving one of the two metrics causes worse results in the other category. The graph shows that the additional training epochs improved the overall detection performance. Additionally, we see that the results initially became worse when the optimizer state was lost. The checkpoints created by Keras do not include the optimizer state. However, it recovered as the training continued.

Next, Figure 5.7 compares the precision, recall, and associated F1 score. Furthermore, it plots all three metrics against the corresponding thresholds. The best F1 score achieved by the model is 0.5409. It achieves this result when using a threshold of 0.0263. This threshold optimizes the ratio between correctly detected anomalies and false positives generated.

Figure 5.8 shows the ROC curve of the three models. It includes the AUC, which the literature typically uses to compare models. The ROC curves support the conclusions drawn from Figure 5.6. The additional epochs improved the model's detection performance.

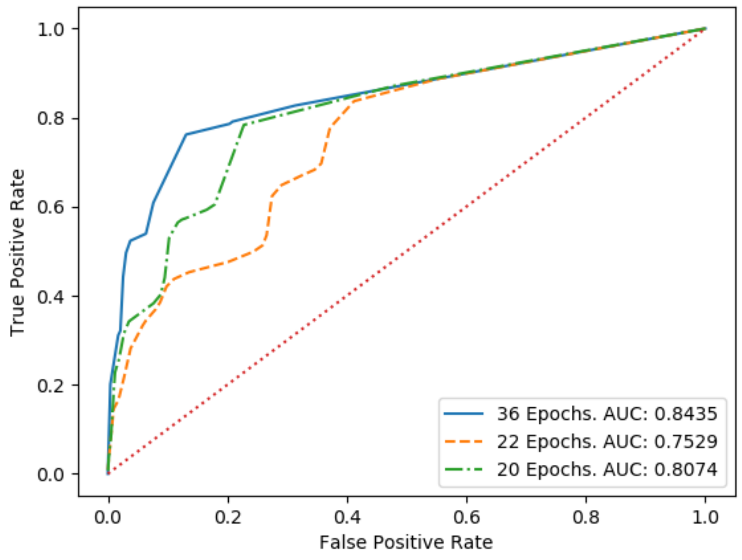

Figure 5.8.: ROC curve and AUC of the model.

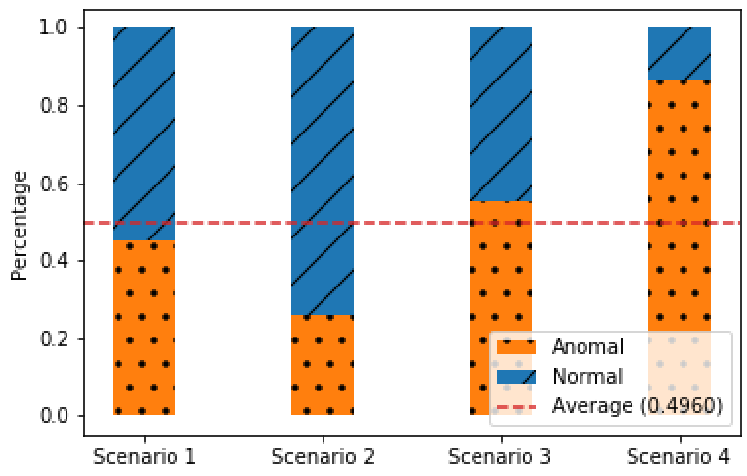

Figure 5.9.: Detection performance per scenario.

Finally, we show the model's performance on the four defined scenarios. We use the best-performing threshold identified in Figure 5.7. The used threshold is 0.0263. Figure 5.9 shows the results for each scenario. The model performs best in detecting anomalies of the fourth scenario. In the fourth scenario, we randomly removed requests from the trace. Our system classifies 86.48% of traces correctly as anomalous. In contrast, it only detects 25.80% of traces from scenario two as anomalies. Scenario two assumes different behavior on weekdays and weekends. For the scenario based on browser and device class changes, the model correctly detects 45.31% of samples. Changing the time difference between requests caused anomalous behavior in 55.34% of the cases.

5.3.3. Discussion

The results show that the system successfully learned to predict user behavior. If actions do not match the learned model, the system is capable of detecting these divergences.

Determining the optimal threshold depends on the acceptable number of false positives. Any anomaly detection system loses acceptance rapidly when raising too many false positives. Hence, we must accept to miss some attacks. Nevertheless, any prevented attack is valuable. Which ratio of false positives is too high cannot be defined in general. It depends on the organization running the anomaly detection system.

One common practice for threshold selection is to base it on the F1 score. The F1 score is the harmonic mean of precision and recall. For our work, we manually selected the F1 score and the corresponding threshold. We selected the highest F1 score. There was no other combination of precision and recall that performed comparably well. However, other case studies may encounter a situation where different configurations achieve similar F1 scores. We leave the issue of automated threshold selection open for further research.

The breakdown of the detection performance for each scenario showed different detection rates. The system missed many anomalies from the second scenario. In this scenario, we swapped working day and weekend requests. We note that we suspect this scenario to depend heavily on the origin of the data. Applications used in business contexts may expose higher differences between working days and weekends.

The extend of the performance difference of the models after epoch 22 and 36 was unexpected. The loss and accuracy reported during training suggested a smaller divergence between the two. This behavior was likely caused by the loss of the optimizer state when restarting the training. Nevertheless, the unexpected difference shows a common pitfall in machine learning projects. It is often hard for humans to interpret the numbers correctly. This highlights the need for further research in the area of *explainable AI*. Explainable AI tries to reason about decisions of machine learning algorithms.

When working on the best-performing architecture, we encountered promising architecture building blocks. Some of them did not deliver the expected results. In the next section, we will discuss some noteworthy observations of elements that did not work in the context of this work.

5.4. Underperforming Architecture Variants

We evaluated a variety of promising structural elements while developing our architecture. Not all of them performed well enough to be included in the final version. In this section, we highlight some of the lessons learned during this process.

First, we evaluated 16, 20, and 50-dimensional embeddings inspired by architectures related to our work [LZY17; Tuo+17; Yua+18]. However, their architectures do not follow the Autoencoder pattern. We applied the embedding to the text-based features only. We evaluated character-based and word-based embeddings. It turned out that training them from scratch is too costly.

Khazan [Kha16] gives the following definition for embeddings: "Like an [A]utoencoder, this type of model learns a vector space embedding for some data." Hence, we tested a combination by adding pre-trained word embeddings like Word2Vec [Mik+13], FastText [Jou+17; Boj+17], and GloVec [PSM14] to our architecture. In theory, re-using embeddings reduces the amount of training required. The pre-trained embeddings we tested construct the word contexts from news articles. However, this did not improve the model's performance. We concluded that word contexts in news articles are too different from word contexts in URLs.

Next, we evaluated several additional layer types during architecture development. However, none of them caught up with the final architecture's performance. For example, we tried CNN and TCN layers. He and Zhao [HZ19] show that TCNs can detect anomalies in time series. After combining the TCN layers with Autoencoders, we could not observe a satisfying performance anymore. In theory, CNNs should be capable of learning to reduce requests that are semantically equivalent but use different orders of GET parameters to the same compressed representation. However, the overall performance of the system did not support that assumption.

Another architecture variant we tested was one where we substituted all GRU cells with LSTM cells. LSTMs need more resources for training but provide slightly preciser outputs than GRUs. For our architecture, using LSTMs led to computation errors due to vanishing or exploding gradients. The errors happened during the training and can be observed when TensorFlow reports the loss with the special value *Not a Number*. Therefore, the final architecture favors GRUs over LSTMs.

Additionally, we tried to chain GRU layers in the encoder and decoder, respectively. Each GRU down the chain then learns features more abstract than the previous one. The training effort increases with each GRU layer. However, the performance did not improve in a comparable significance. For that reason, we decided not to use chained recurrent layers in our final architecture.

Finally, we tried a decoder structure in the behavior Autoencoder that creates a sequence of interim representations, each similar to the latent space of the request Autoencoder. We then could have used the same layout for the request decoder and the deeper layers of the behavior decoder. By sharing the layer weights and biases, this could have cheapened the training further. However, this did not work as expected, and we had to discard this kind of architecture.

6. Related Work

The list of research areas that interconnect with our work is extensive. In this chapter, we focus on work closely related to the presented approach. For each presented work, we give a summary of the taken approach and discuss strengths and possible opportunities for improvements. We also compare their performance to the performance of our system as far as possible. The criteria for comparability forced us to include work that applies machine learning techniques to log data only.

For the interested reader, who wants a sound context for anomaly detection first, we refer to Chandola, Banerjee, and Kumar [CBK09]. They discuss numerous possible domains of application and include standard techniques established in the respective domain. Chalapathy and Chawla [CC19] provide a comprehensive survey for deep learning-based approaches to anomaly detection.

We organized the chapter into two sections. First, we focus on approaches that work on a per-request basis. These approaches serve as a reference to quantify the performance of the request Autoencoder. Second, we show approaches trying to mine user behavior. We compare those to the performance of the behavior Autoencoder.

6.1. Request-Based Anomaly Detection

This section focuses on work that identifies single outlying requests. The approaches do not target behavioral outliers but focus on single, invalid requests. We, in contrast, detect requests that can be valid, but simply are not expected from the given user at that moment. At first sight, the two kinds of anomaly seem different, but many concepts used in the presented papers in this section are applicable to the request Autoencoder.

Cao, Qiao, and Lyu [CQL17] developed an anomaly detection system building upon HMMs. The HMM learns transitions between individual tokens of the HTTP request and not transitions between requests. They use a private dataset containing 4,690,000 HTTP requests stored in a standard log format. The system achieves an accuracy of 93.54%, a precision of 92.07%, a TPR of 89.90%, and an FPR of 4.09%. However, to scale to larger applications, the system needs significant amounts of resources. It has to train a separate model for each parameter of each request. This implies that the system must know all requests in advance. In contrast, our model does not expose this requirement in general.

Liang, Zhao, and Ye [LZY17] model the sequential information present in HTTP requests. Their approach is similar to the approach presented by Cao, Qiao, and Lyu [CQL17]. They use word-based tokenization. Based on the sequence of tokens, they train several sequence-aware neural networks consisting of LSTM and GRU cells. Their system achieves an accuracy between 85.15% and 98.56%, depending on the architecture details. Our request Autoencoder delivers results in the upper third of that region, too. However, we prioritize the behavior Autoencoder over the request Autoencoder. Hence, our approach does not perform as well in the specific domain as theirs does.

Zolotukhin et al. [Zol+14] use a two-stepped approach for the data preprocessing to extract information from HTTP log files. First, they extract per-request information of each line comparable to our approach. Then, they cluster requests into time-based bins. They extract additional features like the number of requests per time interval from these bins. Based on the feature under investigation, they apply different machine learning algorithms depending on which performs best. The developed model achieves an accuracy between 98.29% and 99.24%, depending on the active set of hyperparameters. Finding the best machine learning algorithm on a for each feature is computationally expensive. With our approach, it is not necessary to train multiple classifiers. Despite computational power, the approach presented by Zolotukhin et al. requires expert knowledge of several different machine learning approaches, too.

Dong et al. [Don+18] built a system that is capable of adapting to changing data more quickly than comparable work. The authors trained the system on 112,397 HTTP request logs. Based on these log entries, they trained twelve individual base classifiers. As a final step, they searched the best-performing combination of three out of the twelve base classifiers. The proposed approach works well. The authors report an F1 score of 94.79%. However, similar to the approach presented by Zolotukhin et al. [Zol+14], training twelve classifiers requires an extensive amount of resources. Our approach avoids training multiple classifiers. As a result, it is easier to train because different classifiers require different preprocessing steps. Additionally, the hardware requirements for the techniques are different. Adapting to changing data is an interesting aspect. In the future, it might be worthwhile investigating whether such an approach can also be applied to our solution.

6.2. Behavior-Based Anomaly Detection

Detecting change in user behavior attracted attention recently. Therefore, this section discusses work related to that topic. We discuss some well-performing approaches and compare their performance to ours.

DeepLog [Du+17] calculates the likelihood of all actions. Therefore, it needs to know all possible actions in advance. The system accepts a set of recent actions of the user as input. The authors train an LSTM-based neural network for each parameter of each log line to detect time series-based divergences. As the number of parameters is large for complex systems, their approach increases the required resources for training significantly. DeepLog performs well and achieves a precision rate of 95%, a recall rate of 96%, and an F1 score of 96%. Our system differs from DeepLog, as we only decide if the next request is normal or anomalous. This way, we do not need to know all the log entries in advance. Additionally, we do not train separate models for different parameters.

Yuan et al. [Yua+18] use an HMM-based approach to model user behavior. They partition the log file by user and week. The authors do not base their system on HTTP logs. Instead, they use five classes of activities performed on a computer. *Email*, *HTTP*, *logon/logoff*, *file*, and *device* are the categories that describe the generic structure of the dataset. The developed system performs reasonably well. The authors report that they achieved 94.49% for the AUC of the ROC curve. Since the activity on weekends is significantly different from the activity during weekdays in working environments, the authors decided to exclude weekends from their model. The approach presented by Yuan et al. shows that user behavior models based on log files can be learned reasonably well. We built on this approach, and show that HTTP log files are also suitable for learning user behavior.

Lu and Wong [LW19] not only detect anomalies but try to classify the kind of insider threat, too. The authors used the same dataset as Yuan et al. [Yua+18] did. The dataset consists of general events performed on a computer and not HTTP requests only. They also use a sliding window approach, where the system considers the past 20 requests. They train an LSTM-based neural network, which is similar in structure to the one used by Du et al. [Du+17]. The results achieved with the system are good. The authors report a precision rate of 73%, a recall rate of 91%, and an F1 score of 81%. The dataset used by Lu and Wong is significantly different from the one we used. They collect the data they need locally on the users' computers. In contrast, our approach collects the log entries server-side. First, this avoids administrative overhead. Second, malicious users could intercept the data they transmit for analysis and hide their activities. With our setup, users cannot influence the data recorded for the anomaly detection system. However, their work emphasizes that neural networks are capable of learning user behavior.

The abstract structure Brown et al. [Bro+18] use for their anomaly detection system is similar to ours. However, they use LSTM-based networks, whereas we use Autoencoders that include recurrent elements. Brown et al. first transform the input sequence into a compressed representation. It is then passed to a context LSTM to model user behavior. Their work does not use HTTP request logs but instead uses network connection data consisting of eight features. They also compared char-based and word-based tokenization. The developed system achieves an AUC ratio for the ROC curve between 96.3% and 99.2%, depending on the configuration used. While the dataset they used is not comparable to ours, the work shows that the approach to first compress requests and then extract user behavior works.

Tuor et al. [Tuo+17] created a system that learns user behavior and adapts to changes over time. The authors selected a more recent version of the dataset used by works presented earlier in this chapter [Yua+18; LW19]. Tuor et al. included additional meta-information provided by the dataset like *role*, *department*, and *supervisor*. Like our system, their system outputs anomalies. Experts must decide if the anomalies are malicious. They base their system on a daily budget that defines how many requests the experts can investigate. Additionally, they assume that the experts always decide correctly whether an anomaly is malicious. The budget size influences the recall rate significantly. Since we are not using a budget-based approach, the systems are hardly comparable. Nevertheless, they used architectural styles that influenced our architecture-related decisions.

7. Conclusion

Modern software relies on Application Programming Interfaces (APIs) as a central access point to data. The global interconnection in private and industrial contexts requires exposing APIs to the Internet. Therefore, APIs need robust protection mechanisms. Existing techniques perform well in protecting against attacks with anomalous payloads. However, they can not detect whether the current behavioral pattern is atypical for a given user.

Describing behavioral patterns with traditional approaches is cumbersome and difficult. For this reason, machine learning became popular to tackle the problem. Additionally, modern systems handle large amounts of traffic. Machine learning is designed to work with large amounts of data, which makes it perfectly suitable.

In this thesis, we presented a novel approach to detect divergences from behavioral patterns. The system first learns the patterns typical for users of an API. It uses HTTP log files from web servers as the data source. Comparable other work uses full HTTP requests. Log files do not include most headers and the body of HTTP requests. Hence, our system has less information available to extract behavioral patterns. One of the major contributions of this thesis, therefore, was evaluating whether a log entry contains sufficient information.

The proposed system uses a Deep Neural Network (DNN) operating in two logical steps. First, an Autoencoder compresses the requests to a latent space. Then, a second Autoencoder models the behavior of users based on that compressed representation. Related work suggested this approach for differently structured datasets. Compressing the input first widens the options for layers learning behavioral patterns. With this thesis, we contribute a verification that the two-stepped approach works for log file entries.

We evaluate the performance using two metrics. First, we show the model's generalization capability. We define a threshold being above 99.90% of the reconstruction errors from the training samples. Applied to previously unseen data, 99.94% of the samples were below that threshold. Second, we use a set of generated anomalies to verify the model's detection capability. We showed that the defined threshold for anomalies plays a crucial role. The highest F1 score our system achieves is 0.5409. It achieves an Area Under Curve (AUC) ratio for the Receiver Operating Characteristic (ROC) curve of 84.35%.

Our thorough evaluation shows that Convolutional Neural Networks (CNNs) and Temporal Convolutional Networks (TCNs) are not suitable for our task. The initial hypothesis that they improve the quality of the compressed request representations could not be confirmed. Instead, we used Gated Recurrent Units (GRUs) cells in combination with attention mechanisms.

To summarize, we showed that an Autoencoder-based architecture in combination with GRUs and attention mechanisms are capable of learning user behavior. While it can detect behavior changes, a configuration that allows the system to detect all deviating patterns produces too many false positives. Hence, a trade-off between precision and recall is necessary. We defined the threshold manually based on the F1 score.

Future work could relate the performance of behavioral outlier detection systems to the number of users. As the number of users increases, a system might perform better if grouping similar users. Another aspect worth inspecting is the handling of benign behavior change over extended periods. Neural networks support online training. Hence, they can adapt in theory. We think it is worthwhile verifying this claim.

Appendix

Appendix A.

Keras Model Summary

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
Input-Embedding-Features (Input [(None, 5, 620)]     0
--------------------------------------------------------------------------------------------------
EB-InputChannel0 (Lambda)       (None, 620)          0           Input-Embedding-Features[0][0]
--------------------------------------------------------------------------------------------------
EB-InputChannel1 (Lambda)       (None, 620)          0           Input-Embedding-Features[0][0]
--------------------------------------------------------------------------------------------------
EB-InputChannel2 (Lambda)       (None, 620)          0           Input-Embedding-Features[0][0]
--------------------------------------------------------------------------------------------------
EB-InputChannel3 (Lambda)       (None, 620)          0           Input-Embedding-Features[0][0]
--------------------------------------------------------------------------------------------------
EB-InputChannel4 (Lambda)       (None, 620)          0           Input-Embedding-Features[0][0]
--------------------------------------------------------------------------------------------------
dense (Dense)                   (None, 310)          192510      EB-InputChannel0[0][0]
                                                                 EB-InputChannel1[0][0]
                                                                 EB-InputChannel2[0][0]
                                                                 EB-InputChannel3[0][0]
                                                                 EB-InputChannel4[0][0]
--------------------------------------------------------------------------------------------------
reshape (Reshape)               (None, 310, 1)       0           dense[0][0]
                                                                 dense[1][0]
                                                                 dense[2][0]
                                                                 dense[3][0]
                                                                 dense[4][0]
--------------------------------------------------------------------------------------------------
Input-Meta-Features (InputLayer [(None, 5, 30)]      0
--------------------------------------------------------------------------------------------------
bidirectional (Bidirectional)   (None, 128)          25728       reshape[0][0]
                                                                 reshape[1][0]
                                                                 reshape[2][0]
                                                                 reshape[3][0]
                                                                 reshape[4][0]
--------------------------------------------------------------------------------------------------
MF-InputChannel0 (Lambda)       (None, 30)           0           Input-Meta-Features[0][0]
--------------------------------------------------------------------------------------------------
MF-InputChannel1 (Lambda)       (None, 30)           0           Input-Meta-Features[0][0]
--------------------------------------------------------------------------------------------------
MF-InputChannel2 (Lambda)       (None, 30)           0           Input-Meta-Features[0][0]
--------------------------------------------------------------------------------------------------
MF-InputChannel3 (Lambda)       (None, 30)           0           Input-Meta-Features[0][0]
--------------------------------------------------------------------------------------------------
MF-InputChannel4 (Lambda)       (None, 30)           0           Input-Meta-Features[0][0]
--------------------------------------------------------------------------------------------------
Compact-Request-w-MetaInfo (Con (None, 158)          0           bidirectional[0][0]
                                                                 MF-InputChannel0[0][0]
                                                                 bidirectional[1][0]
                                                                 MF-InputChannel1[0][0]
                                                                 bidirectional[2][0]
                                                                 MF-InputChannel2[0][0]
                                                                 bidirectional[3][0]
                                                                 MF-InputChannel3[0][0]
                                                                 bidirectional[4][0]
                                                                 MF-InputChannel4[0][0]
__________________________________________________________________________________________________
Latent-Space-Request-AE (Dense) (None, 96)          15264       Compact-Request-w-MetaInfo[0][0]
                                                                 Compact-Request-w-MetaInfo[1][0]
                                                                 Compact-Request-w-MetaInfo[2][0]
                                                                 Compact-Request-w-MetaInfo[3][0]
                                                                 Compact-Request-w-MetaInfo[4][0]
__________________________________________________________________________________________________
```

Listing A.1: Keras model summary of the request encoder.

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
dense_2 (Dense)                 (None, 126)          12222       Latent-Space-Request-AE[0][0]
                                                                 Latent-Space-Request-AE[1][0]
                                                                 Latent-Space-Request-AE[2][0]
                                                                 Latent-Space-Request-AE[3][0]
                                                                 Latent-Space-Request-AE[4][0]
__________________________________________________________________________________________________
dense_3 (Dense)                 (None, 620)          78740       dense_2[0][0]
                                                                 dense_2[1][0]
                                                                 dense_2[2][0]
                                                                 dense_2[3][0]
                                                                 dense_2[4][0]
__________________________________________________________________________________________________
reshape_2 (Reshape)             (None, 620, 1)       0           dense_3[0][0]
                                                                 dense_3[1][0]
                                                                 dense_3[2][0]
                                                                 dense_3[3][0]
                                                                 dense_3[4][0]
__________________________________________________________________________________________________
dense_5 (Dense)                 (None, 200)          25400       dense_2[0][0]
                                                                 dense_2[1][0]
                                                                 dense_2[2][0]
                                                                 dense_2[3][0]
                                                                 dense_2[4][0]
__________________________________________________________________________________________________
bidirectional_3 (Bidirectional) (None, 620, 64)      6720        reshape_2[0][0]
                                                                 reshape_2[1][0]
                                                                 reshape_2[2][0]
                                                                 reshape_2[3][0]
                                                                 reshape_2[4][0]
__________________________________________________________________________________________________
dropout_3 (Dropout)             (None, 200)          0           dense_5[0][0]
                                                                 dense_5[1][0]
                                                                 dense_5[2][0]
                                                                 dense_5[3][0]
                                                                 dense_5[4][0]
__________________________________________________________________________________________________
dense_4 (Dense)                 (None, 620, 235)     15275       bidirectional_3[0][0]
                                                                 bidirectional_3[1][0]
                                                                 bidirectional_3[2][0]
                                                                 bidirectional_3[3][0]
                                                                 bidirectional_3[4][0]
__________________________________________________________________________________________________
dense_6 (Dense)                 (None, 30)           6030        dropout_3[0][0]
                                                                 dropout_3[1][0]
                                                                 dropout_3[2][0]
                                                                 dropout_3[3][0]
                                                                 dropout_3[4][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 3100, 235)    0           dense_4[0][0]
                                                                 dense_4[1][0]
                                                                 dense_4[2][0]
                                                                 dense_4[3][0]
                                                                 dense_4[4][0]
__________________________________________________________________________________________________
concatenate_2 (Concatenate)     (None, 150)          0           dense_6[0][0]
                                                                 dense_6[1][0]
                                                                 dense_6[2][0]
                                                                 dense_6[3][0]
```

```
                                                                         dense_6[4][0]
 ____________________________________________________________________________________________
O-Requests-Decoder (Reshape)    (None, 5, 620, 235)  0           concatenate_1[0][0]
 ____________________________________________________________________________________________
O-Requests-Decoder-Meta (Reshap (None, 5, 30)        0           concatenate_2[0][0]
 ____________________________________________________________________________________________
```

Listing A.2: Keras model summary of the request decoder.

```
Layer (type)                    Output Shape         Param #     Connected to
==============================================================================================
Latent-Space-Request-AE (Dense) (None, 96)           15264       Compact-Request-w-MetaInfo[0][0]
                                                                 Compact-Request-w-MetaInfo[1][0]
                                                                 Compact-Request-w-MetaInfo[2][0]
                                                                 Compact-Request-w-MetaInfo[3][0]
                                                                 Compact-Request-w-MetaInfo[4][0]
 ____________________________________________________________________________________________
concatenate (Concatenate)       (None, 480)          0           Latent-Space-Request-AE[0][0]
                                                                 Latent-Space-Request-AE[1][0]
                                                                 Latent-Space-Request-AE[2][0]
                                                                 Latent-Space-Request-AE[3][0]
                                                                 Latent-Space-Request-AE[4][0]
 ____________________________________________________________________________________________
reshape_1 (Reshape)             (None, 5, 96)        0           concatenate[0][0]
 ____________________________________________________________________________________________
bidirectional_2 (Bidirectional) (None, 5, 128)       62208       reshape_1[0][0]
 ____________________________________________________________________________________________
Latent-Space-Behavior-Autoencod (None, 128)          16640       bidirectional_2[0][0]
 ____________________________________________________________________________________________
```

Listing A.3: Keras model summary of the behavior encoder.

```
Layer (type)                    Output Shape         Param #     Connected to
==============================================================================================
repeat_vector (RepeatVector)    (None, 5, 128)       0           Latent-Space-Behavior-Autoencoder
 ____________________________________________________________________________________________
gru_3 (GRU)                     (None, 5, 620)       1395000     repeat_vector[0][0]
 ____________________________________________________________________________________________
dense_1 (Dense)                 (None, 60)           7740        Latent-Space-Behavior-Autoencoder
 ____________________________________________________________________________________________
time_distributed (TimeDistribut (None, 5, 235, 620)  0           gru_3[0][0]
 ____________________________________________________________________________________________
dropout_1 (Dropout)             (None, 60)           0           dense_1[0][0]
 ____________________________________________________________________________________________
permute (Permute)               (None, 5, 620, 235)  0           time_distributed[0][0]
 ____________________________________________________________________________________________
repeat_vector_2 (RepeatVector)  (None, 5, 60)        0           dropout_1[0][0]
 ____________________________________________________________________________________________
O-Behavior-Decoder (Dense)      (None, 5, 620, 235)  55460       permute[0][0]
 ____________________________________________________________________________________________
O-Behavior-Decoder-Meta (Dense) (None, 5, 30)        1830        repeat_vector_2[0][0]
 ____________________________________________________________________________________________
```

Listing A.4: Keras model summary of the behavior decoder.

Appendix B.

Custom Keras Layer: AttentionWithContext

```python
from tensorflow.keras import backend as K
from tensorflow.keras import initializers, regularizers, constraints
from tensorflow.keras.layers import Layer

def dot_product(x, kernel):
    return K.squeeze(K.dot(x, K.expand_dims(kernel)), axis=-1)

class AttentionWithContext(Layer):
    """
    Attention operation, with a context/query vector, for temporal data.
    Supports Masking.
    Follows the work of Yang et al. [Yan+16b]
    by using a context vector to assist the attention
    # Input shape
        3D tensor with shape: `(samples, steps, features)`.
    # Output shape
        2D tensor with shape: `(samples, features)`.
    How to use:
    Just put it on top of an RNN Layer (GRU/LSTM/SimpleRNN)
    Set return_sequences=True.
    The dimensions are inferred based on the output shape of the RNN.
    Note: The layer has been tested with Keras 2.0.6
    Example:
        model.add(LSTM(64, return_sequences=True))
        model.add(AttentionWithContext())
        # next add a Dense layer (for classification/regression) or whatever...
    """

    def __init__(self,
                 W_regularizer=None, u_regularizer=None, b_regularizer=None,
                 W_constraint=None, u_constraint=None, b_constraint=None,
                 bias=True, **kwargs):

        self.init = initializers.get('glorot_uniform')

        self.W_regularizer = regularizers.get(W_regularizer)
        self.u_regularizer = regularizers.get(u_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)

        self.W_constraint = constraints.get(W_constraint)
        self.u_constraint = constraints.get(u_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        super(AttentionWithContext, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3

        self.W = self.add_weight(shape=(input_shape[-1], input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        if self.bias:
            self.b = self.add_weight(shape=(input_shape[-1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)

        self.u = self.add_weight(shape=(input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_u'.format(self.name),
                                 regularizer=self.u_regularizer,
                                 constraint=self.u_constraint)

        super(AttentionWithContext, self).build(input_shape)

    def compute_mask(self, input, input_mask=None):
        # do not pass the mask to the next layers
        return None

    def call(self, x, mask=None):
        uit = dot_product(x, self.W)

        if self.bias:
            uit += self.b

        uit = K.tanh(uit)
        ait = dot_product(uit, self.u)

        a = K.exp(ait)

        # apply mask after the exp. will be re-normalized next
        if mask is not None:
            # Cast the mask to floatX to avoid float64 upcasting in theano
            a *= K.cast(mask, K.floatx())

        # in some cases especially in the early stages of training
        # the sum may be almost zero and this results in NaN's.
        # A workaround is to add a very small positive number eps to the sum.
        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(), K.floatx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0], input_shape[-1]

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'bias': self.bias,
            'b_constraint': self.b_constraint,
            'u_constraint': self.u_constraint,
            'W_constraint': self.W_constraint,
            'b_regularizer': self.b_regularizer,
            'u_regularizer': self.u_regularizer,
            'W_regularizer': self.W_regularizer,
        })
        return config
```

Listing B.1: Implementation of the attention mechanism Yang et al. [Yan+16b] proposed. Code initially released under Apache 2.0 license by User Kepler456b [Use19]. Minor modifications applied.

Bibliography

[AH18] Nancy Agarwal and Syed Zeeshan Hussain. "A Closer Look at Intrusion Detection System for Web Applications." In: *Security and Communication Networks* 2018 (2018). ISSN: 1939-0122. DOI: 10.1155/2018/9601357. arXiv: 1803.06153 (cit. on p. 7).

[Alo+19] Md Zahangir Alom et al. "A state-of-the-art survey on deep learning theory and architectures." In: *Electronics* 8.3 (2019), p. 292. ISSN: 20799292. DOI: 10.3390/electronics8030292 (cit. on p. 27).

[BB12] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305. ISSN: 15324435 (cit. on pp. 15, 16).

[BCB15] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." In: *3rd International Conference on Learning Representations (ICLR)*. 2015. arXiv: 1409.0473 (cit. on pp. 31, 33).

[Ben+13] Yoshua Bengio et al. "Generalized denoising auto-encoders as generative models." In: *Advances in Neural Information Processing Systems* (2013), pp. 1–9. ISSN: 10495258. arXiv: 1305.6663 (cit. on p. 29).

[BH89] Pierre Baldi and Kurt Hornik. "Neural networks and principal component analysis: Learning from examples without local minima." In: *Neural Networks* 2.1 (1989), pp. 53–58. ISSN: 08936080. DOI: 10.1016/0893-6080(89)90014-2 (cit. on p. 28).

[BKK18] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling." In: *arXiv preprint* (2018). arXiv: 1803.01271 (cit. on pp. 32, 33).

[Boj+17] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information." In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00051. arXiv: 1607.04606 (cit. on pp. 12, 77).

[Bro+18] Andy Brown et al. "Recurrent Neural Network Attention Mechanisms for Interpretable System Log Anomaly Detection." In: *Proceedings of the First Workshop on Machine Learning for Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 9781450358651. DOI: 10.1145/3217871.3217872 (cit. on p. 82).

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey." In: *ACM Comput. Surv.* 41.July (2009), 15:1–15:58. ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. arXiv: arXiv:1011.1669v3 (cit. on pp. 8, 79).

[CC19] Raghavendra Chalapathy and Sanjay Chawla. "Deep Learning for Anomaly Detection: A Survey." In: *arXiv preprint* (2019). arXiv: 1901.03407 (cit. on pp. 35, 79).

[Cho+14] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." In: *Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference (EMNLP)* (2014), pp. 1724–1734. DOI: 10.3115/v1/d14-1179. arXiv: 1406.1078 (cit. on pp. 26, 33).

[Cho18] Francois Chollet. *Deep Learning with Python*. 1st. New York, NY: Manning Publications Co., 2018, p. 361. ISBN: 9780996452762 (cit. on pp. 9, 15, 18, 22, 24).

[CQL17] Qimin Cao, Yinrong Qiao, and Zhong Lyu. "Machine learning to detect anomalies in web log analysis." In: *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. 2017, pp. 519–523. ISBN: 9781509063505. DOI: 10.1109/CompComm.2017.8322600 (cit. on pp. 7, 80).

[CV95] Corinna Cortes and Vladimir Vapnik. "Support Vector Machine." In: *Machine Learning* 20.3 (1995), pp. 273–297 (cit. on p. 10).

[Den87] Dorothy E Denning. “An Intrusion-Detection Model.” In: *IEEE Transactions on Software Engineering* SE-13.2 (1987), pp. 222–232 (cit. on p. 2).

[Det19] Tim Dettmers. *No Which GPU(s) to Get for Deep Learning: My Experience and Advice for Using GPUs in Deep Learning*. Apr. 2019. URL: https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/ (visited on 03/05/2020) (cit. on p. 61).

[Don+18] Ying Dong et al. “An adaptive system for detecting malicious queries in web attacks.” In: *Science China Information Sciences* 61.3 (2018). ISSN: 18691919. DOI: 10.1007/s11432-017-9288-4. arXiv: 1701.07774 (cit. on p. 81).

[Du+17] Min Du et al. “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning.” In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1285–1298. ISBN: 9781450349468. DOI: 10.1145/3133956.3134015 (cit. on pp. 4, 81, 82).

[FD19] Osama Faker and Erdogan Dogdu. “Intrusion detection using big data and deep learning techniques.” In: *ACMSE 2019 - Proceedings of the 2019 ACM Southeast Conference* (2019), pp. 86–93. DOI: 10.1145/3299815.3314439 (cit. on p. 7).

[FT00] Roy T Fielding and Richard N Taylor. “Architectural styles and the design of network-based software architectures.” PhD thesis. University of California, Irvine Irvine, 2000 (cit. on p. 27).

[Gao+18] Peng Gao et al. “SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection.” In: *27th USENIX Security Symposium* (2018), pp. 639–656 (cit. on p. 9).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN: 9780262035613 (cit. on pp. 10, 12–15, 18, 19, 22, 29).

[GD98] M. W. Gardner and S. R. Dorling. “Artificial neural networks (the multilayer perceptron) - a review of applications in the atmospheric sciences.” In: *Atmospheric Environment* 32.14-15 (1998), pp. 2627–2636. ISSN: 13522310. DOI: 10.1016/S1352-2310(97)00447-0 (cit. on pp. 14, 23, 33).

[Gér19] Aurélien Géron. *Hands-on Machine Learning with Sklearn, keras & tensorflow*. 2nd. O'Reilly Media, 2019. ISBN: 9781492032649 (cit. on p. 61).

[GLH15] Salvador García, Julián Luengo, and Francisco Herrera. *Data preprocessing in data mining*. Springer International Publishing, 2015. ISBN: 9783319102467. DOI: 10.1007/978-3-319-10247-4 (cit. on pp. 10–12).

[Goo20] Google. *GPUs on Compute Engine*. Mar. 2020. URL: https://cloud.google.com/compute/docs/gpus/ (visited on 04/01/2020) (cit. on p. 61).

[GS00] Felix A. Gers and Jürgen Schmidhuber. "Recurrent Nets that Time and Count." In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*. 2000, pp. 189–194 (cit. on p. 26).

[Hay05] Simon Haykin. *Neural Networks - A Comprehensive Foundation*. 9th ed. Prentice Hall PTR, 2005, p. 823. ISBN: 81-7808-300-0 (cit. on p. 23).

[He+16] Kaiming He et al. "Deep residual learning for image recognition." In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2016-Decem (2016), pp. 770–778. ISSN: 10636919. DOI: 10.1109/CVPR.2016.90. arXiv: 1512.03385 (cit. on p. 21).

[Hoc91] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen." In: *Diploma, Technische Universität München* (1991), p. 66 (cit. on p. 21).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735 (cit. on pp. 21, 26, 33).

[HZ19] Yangdong He and Jiabao Zhao. "Temporal Convolutional Networks for Anomaly Detection in Time Series." In: *Journal of Physics: Conference Series*. Vol. 1213. 4. IOP Publishing, 2019. DOI: 10.1088/1742-6596/1213/4/042050 (cit. on p. 78).

[Jad+17] Max Jaderberg et al. “Population Based Training of Neural Networks.” In: *arXiv preprint* (2017). arXiv: 1711.09846 (cit. on pp. 15, 16).

[Jia+16] Yongyao Jiang et al. “Reconstructing sessions from data discovery and access logs to build a semantic knowledge base for improving data discovery.” In: *ISPRS International Journal of Geo-Information* 5.5 (2016). ISSN: 22209964. DOI: 10.3390/ijgi5050054 (cit. on p. 53).

[Jou+17] Armand Joulin et al. “Bag of tricks for efficient text classification.” In: *15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017 - Proceedings of Conference* 2 (2017), pp. 427–431. DOI: 10.18653/v1/e17-2068. arXiv: 1607.01759 (cit. on pp. 12, 77).

[KB14] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” In: (2014), pp. 1–15. arXiv: arXiv:1412.6980v9 (cit. on pp. 20, 46).

[KET17] Philipp Kreimel, Oliver Eigner, and Paul Tavolato. “Anomaly-Based Detection and Classification of Attacks in Cyber-Physical Systems.” In: *Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES '17* (2017), pp. 1–6. DOI: 10.1145/3098954.3103155 (cit. on p. 9).

[Kha+19] Asifullah Khan et al. “A Survey of the Recent Architectures of Deep Convolutional Neural Networks.” In: *arXiv preprint* (2019). arXiv: 1901.06032 (cit. on p. 24).

[Kha16] Lenny Khazan. *Lenny #2: Autoencoders and Word Embeddings*. Apr. 2016. URL: https://ayearofai.com/lenny-2-autoencoders-and-word-embeddings-oh-my-576403b0113a (visited on 04/09/2020) (cit. on p. 77).

[Khr+19] Ansam Khraisat et al. “Survey of intrusion detection systems: techniques, datasets and challenges.” In: *Cybersecurity* 2.1 (2019). ISSN: 2523-3246. DOI: 10.1186/s42400-019-0038-7 (cit. on pp. 7, 32).

[KPC15] Poonam Sinai Kenkre, Anusha Pai, and Louella Colaco. "Real Time Intrusion Detection and Prevention System." In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Vol. 327. Cham: Springer International Publishing, 2015, pp. 405–411. ISBN: 978-3-319-11932-8. DOI: 10.1007/978-3-319-11933-5 (cit. on p. 7).

[KS96] Ben Kröse and Patrick van der Smagt. *An introduction to neural networks*. 8th. 1996, p. 135 (cit. on p. 17).

[KW13] Diederik P. Kingma and Max Welling. "Auto-encoding variational bayes." In: *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings* Ml (2013), pp. 1–14. arXiv: 1312.6114 (cit. on p. 30).

[Lar+16] Anders Boesen Lindbo Larsen et al. "Autoencoding beyond pixels using a learned similarity metric." In: *33rd International Conference on Machine Learning, ICML 2016* 4 (2016), pp. 2341–2349. arXiv: 1512.09300 (cit. on p. 30).

[LBH15] Yann André Lecun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 14764687. DOI: 10.1038/nature14539 (cit. on p. 21).

[LeC+98] Yann André LeCun et al. "Efficient backprop." In: *Neural Networks: Tricks of the Trade*. Berlin, Heidelberg: Springer, 1998, pp. 9–50 (cit. on p. 18).

[LeC89] Yann André LeCun. "Generalization and network design strategies." In: *Connectionism in perspective* 19 (1989), pp. 143–155 (cit. on pp. 24, 33).

[Li+19] Yang Li et al. "Disentangled Variational Auto-Encoder for semi-supervised learning." In: *Information Sciences* 482 (2019), pp. 73–85. ISSN: 00200255. DOI: 10.1016/j.ins.2018.12.057. arXiv: 1709.05047 (cit. on p. 30).

[Lia+13] Hung Jen Liao et al. "Intrusion detection system: A comprehensive review." In: *Journal of Network and Computer Applications* 36.1 (2013), pp. 16–24. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2012.09.004 (cit. on pp. 7, 32).

[Liu+17] Weibo Liu et al. "A survey of deep neural network architectures and their applications." In: *Neurocomputing* 234 (2017), pp. 11–26. ISSN: 18728286. DOI: 10.1016/j.neucom.2016.12.038 (cit. on pp. 24, 25).

[LNP19] Saïd Ladjal, Alasdair Newson, and Chi-Hieu Pham. "A PCA-like Autoencoder." In: *arXiv preprint* (2019). arXiv: 1904.01277 (cit. on p. 28).

[LW19] Jiuming Lu and Raymond K. Wong. "Insider Threat Detection with Long Short-Term Memory." In: *ACM International Conference Proceeding Series* (2019). DOI: 10.1145/3290688.3290692 (cit. on pp. 82, 83).

[LZY17] Jingxi Liang, Wen Zhao, and Wei Ye. "Anomaly-Based Web Attack Detection: A Deep Learning Approach." In: *Proceedings of the 2017 VI International Conference on Network, Communication and Computing*. ICNCC 2017 (2017), pp. 80–85. DOI: 10.1145/3171592.3171594 (cit. on pp. 77, 80).

[Mac+18] Hieu Mac et al. "Detecting Attacks on Web Applications Using Autoencoder." In: *Proceedings of the Ninth International Symposium on Information and Communication Technology* (2018), pp. 416–421. DOI: 10.1145/3287921.3287946 (cit. on p. 2).

[Mik+13] Tomas Mikolov et al. "Efficient estimation of word representations in vector space." In: *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings* (2013), pp. 1–12. arXiv: 1301.3781 (cit. on pp. 12, 77).

[Naz+08] Jamal M Nazzal et al. "Multilayer Perceptron Neural Network (MLPs) For Analyzing the Propoerties of Jordan Oil Shale." In: *World Applied Sciences Journal* 5.5 (2008), pp. 546–552. ISSN: 1818-4952 (cit. on p. 23).

[NH10] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines." In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814. ISBN: 9781605589077 (cit. on p. 18).

[NS08] Arvind Narayanan and Vitaly Shmatikov. “Robust de-anonymization of large sparse datasets.” In: *Proceedings - IEEE Symposium on Security and Privacy* (2008), pp. 111–125. ISSN: 10816011. DOI: 10.1109/SP.2008.33 (cit. on p. 48).

[Ola15] Christopher Olah. *Understanding LSTM Networks*. Aug. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 02/10/2020) (cit. on p. 27).

[PG17] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly Media, Inc., 2017, p. 532. ISBN: 978-1491914250 (cit. on pp. 15, 20, 24).

[PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training Recurrent Neural Networks.” In: *Proceedings of the 30th International Conference on Machine Learning*. Atlanta, GA, USA: JMLR.org, 2013, pp. 1310–1318. arXiv: 1211.5063 (cit. on p. 22).

[Pou+18] Samira Pouyanfar et al. “A survey on deep learning: Algorithms, techniques, and applications.” In: *ACM Computing Surveys* 51.5 (2018). ISSN: 15577341. DOI: 10.1145/3234150 (cit. on p. 25).

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation.” In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162 (cit. on pp. 12, 77).

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors.” In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 00280836. DOI: 10.1038/323533a0 (cit. on pp. 20, 26, 33).

[Rif+11] Salah Rifai et al. “Contractive auto-encoders: Explicit invariance during feature extraction.” In: *Proceedings of the 28th International Conference on Machine Learning, ICML 2011* 1 (2011), pp. 833–840 (cit. on p. 29).

[Rin+19] Markus Ring et al. "A survey of network-based intrusion detection data sets." In: *Computers and Security* 86 (2019), pp. 147–167. ISSN: 01674048. DOI: 10.1016/j.cose.2019.06.005. arXiv: 1903.02460 (cit. on p. 49).

[RJ86] L. Rabiner and B. Juang. "An introduction to hidden Markov models." In: *IEEE ASSP Magazine* 3.1 (1986), pp. 4–16. DOI: 10.1109/MASSP.1986.1165342 (cit. on p. 10).

[RM19] S. Raschka and V. Mirjalili. *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow* 2. Packt Publishing, 2019. ISBN: 9781789955750 (cit. on p. 60).

[RMW14] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. "Stochastic backpropagation and approximate inference in deep generative models." In: *31st International Conference on Machine Learning, ICML 2014*. Vol. 32. JMLR.org, 2014, pp. 1278–1286. ISBN: 9781634393973. arXiv: 1401.4082 (cit. on p. 30).

[RN95] Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995 (cit. on p. 10).

[Ros+04] Lorenzo Rosasco et al. "Are Loss Functions All the Same?" In: *Neural Computation* 16.5 (2004), pp. 1063–1076. ISSN: 08997667. DOI: 10.1162/089976604773135104 (cit. on p. 19).

[Rud16] Sebastian Ruder. "An overview of gradient descent optimization." In: (2016), pp. 1–14. arXiv: arXiv:1609.04747v2 (cit. on p. 19).

[SB98] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. 1st. The MIT Press, 1998, p. 322. ISBN: 9780262193986 (cit. on p. 10).

[Sha+16] Bobak Shahriari et al. "Taking the human out of the loop: A review of Bayesian optimization." In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. ISSN: 00189219. DOI: 10.1109/JPROC.2015.2494218 (cit. on pp. 15, 16).

[SKP97] Daniel Svozil, Vladimir Kvasnicka, and Jiří Pospíchal. "Introduction to multi-layer feed-forward neural networks." In: *Chemometrics and Intelligent Laboratory Systems* 39 (1997), pp. 43–62. DOI: 10.1016/S0169-7439(97)00061-0 (cit. on pp. 14, 33).

[SLG18] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. "Toward generating a new intrusion detection dataset and intrusion traffic characterization." In: *ICISSP 2018 - Proceedings of the 4th International Conference on Information Systems Security and Privacy*. 2018, pp. 108–116. ISBN: 9789897582820. DOI: 10.5220/0006639801080116 (cit. on p. 49).

[SP97] Mike Schuster and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681. ISSN: 1053587X. DOI: 10.1109/78.650093 (cit. on p. 25).

[Sri+14] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting." In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958. ISSN: 1532-4435 (cit. on pp. 22, 33).

[Sze+15] Christian Szegedy et al. "Going deeper with convolutions." In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2015. ISBN: 9781467369640. DOI: 10.1109/CVPR.2015.7298594. arXiv: 1409.4842 (cit. on p. 21).

[Tec17] TechPowerUp. *NVIDIA Quadro M2200 Mobile Specs*. 2017. URL: https://www.techpowerup.com/gpu-specs/quadro-m2200-mobile.c2922 (visited on 03/21/2020) (cit. on p. 62).

[Tec18a] TechPowerUp. *NVIDIA GeForce RTX 2070 Specs*. 2018. URL: https://www.techpowerup.com/gpu-specs/geforce-rtx-2070.c3252 (visited on 03/21/2020) (cit. on p. 62).

[Tec18b] TechPowerUp. *NVIDIA Tesla T4 Mobile Specs*. 2018. URL: https://www.techpowerup.com/gpu-specs/tesla-t4.c3316 (visited on 03/21/2020) (cit. on p. 62).

[Tuo+17] Aaron Tuor et al. "Deep Learning for Unsupervised Insider Threat Detection in Structured Cybersecurity Data Streams." In: *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*. 2017. ISBN: 978-1577357865. arXiv: 1710.00811 (cit. on pp. 8, 48, 77, 83).

[Use19] User Kepler456b. *GRU with Attention*. 2019. URL: https://www.kaggle.com/isikkuntay/gru-with-attention (visited on 03/24/2020) (cit. on p. 92).

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need." In: *Advances in Neural Information Processing Systems (NIPS)*. 30. 2017, pp. 5998–6008. arXiv: 1706.03762 (cit. on p. 31).

[Vin+19] R. Vinayakumar et al. "Deep Learning Approach for Intelligent Intrusion Detection System." In: *IEEE Access* 7 (2019), pp. 41525–41550. DOI: 10.1109/access.2019.2895334 (cit. on pp. 7, 68).

[Von+08] Luis Von Ahn et al. "reCAPTCHA: Human-based character recognition via web security measures." In: *Science* 321.5895 (2008), pp. 1465–1468. ISSN: 00368075. DOI: 10.1126/science.1160379 (cit. on p. 48).

[WDY19] Mengyue Wu, Heinrich Dinkel, and Kai Yu. "Audio Caption: Listen and Tell." In: *2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2019), pp. 830–834. DOI: 10.1109/icassp.2019.8682377. arXiv: 1902.09254 (cit. on p. 28).

[Wen18] Lilian Weng. *From Autoencoder to Beta-VAE*. Aug. 2018. URL: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html (visited on 02/07/2020) (cit. on p. 30).

[Xu+15] Kelvin Xu et al. "Show, attend and tell: Neural image caption generation with visual attention." In: *32nd International Conference on Machine Learning (ICML)*. Vol. 37. 2015, pp. 2048–2057. ISBN: 9781510810587. arXiv: 1502.03044 (cit. on p. 31).

[YA17] Ozlem Yavanoglu and Murat Aydos. “A review on cyber security datasets for machine learning algorithms.” In: *2017 IEEE International Conference on Big Data (Big Data)*. 2017, pp. 2186–2193. ISBN: 9781538627143. DOI: 10.1109/BigData.2017.8258167 (cit. on p. 49).

[Yan+16a] Xinchen Yan et al. “Attribute2Image: Conditional image generation from visual attributes.” In: *European Conference on Computer Vision*. Springer, 2016, pp. 776–791. ISBN: 9783319464923. DOI: 10.1007/978-3-319-46493-0_47. arXiv: 1512.00570 (cit. on p. 28).

[Yan+16b] Zichao Yang et al. “Hierarchical attention networks for document classification.” In: *2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL HLT 2016* (2016), pp. 1480–1489. DOI: 10.18653/v1/n16-1174 (cit. on pp. 40, 43, 92).

[Yua+18] Fangfang Yuan et al. “Insider Threat Detection with Deep Neural Network.” In: *Computational Science – ICCS 2018*. Ed. by Yong Shi et al. Springer International Publishing, 2018, pp. 43–54. ISBN: 9783319936970. DOI: 10.1007/978-3-319-93698-7_4 (cit. on pp. 8, 77, 81–83).

[Zak19] Farzin Zaker. *Online Shopping Store - Web Server Logs*. Version V1. 2019. DOI: 10.7910/DVN/3QBYB5 (cit. on pp. 3, 50, 51).

[Zol+14] Mikhail Zolotukhin et al. “Analysis of HTTP requests for anomaly detection of web attacks.” In: *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC)*. IEEE, 2014, pp. 406–411. DOI: 10.1109/DASC.2014.79 (cit. on pp. 80, 81).