DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Finding anomalies in software licensing logs using unsupervised methods

ARTEM LOS

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

Finding anomalies in software licensing logs using unsupervised methods

ARTEM LOS

Master in Computer Science
Date: May 28, 2020
Supervisor: Mathias Ekstedt
Examiner: Robert Lagerström
School of Electrical Engineering and Computer Science
Host company: Cryptolens AB
Swedish title: Detektering av anomalier i
mjukvarulicensieringsloggar med oövervakadde metoder

Abstract

Cryptolens is world leading software licensing platform. As a result, it has large amounts of data that is generated when each end user application attempts to verify a license key. Being able to differentiate between normal and anomalous data can provide software vendors with a way to detect fraud and other abnormal behaviour, allowing them to save time on analyzing all the data themselves and increase revenues. It is found that an effective way to find anomalies in software licensing logs is to use the reconstruction error as the anomaly score from either an LSTM or TCN based autoencoder, where the decision boundary is decided by the largest error in the error histogram on the training set.

Sammanfattning

Cryptolens är en världsledande mjukvarulicensieringslösning. Tack vare detta har den stor tillgång till data som är genererad när varje slutanvändare försöker verifiera en licens. Att kunna särskilja mellan normal och anomalisk data ger mjukvaruföretag ett sätt att detektera bedrägerier och annan typ av avvikande användning, som tillåter dem att spara tid på att analysera data:n själva och öka sina intäkter. Det konstateras att ett effektivt sätt att hitta anomalier i mjukvarulicensieringsloggar är genom att använda antingen en autoencoder som bygger på LSTM eller TCN, där beslutsgränsen sätts med hjälp av det största felet i histogrammet som är skapat från träningsdatan.

Contents

Chapter 1

Introduction

An important component when a software company wants to monetize their applications is a software licensing system. Such system is needed because of the following reason. First, it allows them to ensure that license entitlements are enforced, for example, to ensure that their customers can only use the features/modules they have paid for and that if a license has expired, the application will no longer work (more details about these are provided in the *Background*). Secondly, a licensing system helps to automate software delivery when combined with a payment gateway. Oftentimes, a software licensing system can be seen as customer relations management system (CRM) for software vendors (this is especially relevant for B2C sales), as most of the licensing systems provide some degree of customer management. Finally, a licensing system provides a way to gather analytics data about the usage, for example, geographical information, popular features, etc, which can aid in strategical decisions. An illustration of how a licensing system interacts with other systems in shown in Figure 1.1.

In most scenarios, end user application instances will attempt to connect to a license verification server (assuming the application is not completely offline), in order to obtain the recent version of a license. These verifications are a source for log data, which can be analysed to check for possible anomalies. The amount of data available differs significantly depending on the licensing model used and how verifications are set up in the application.

1.1 Why outlier detection is needed

Using the log data from license verifications can allow us to learn the normal behaviour and subsequently detect anomalous behaviour. There are a couple

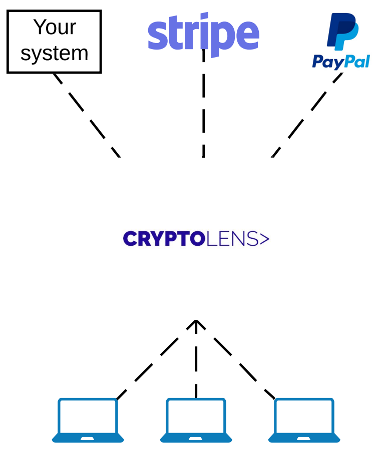

Figure 1.1: An example of how Cryptolens interacts with other systems. Cryptolens ensures that end user application instances can verify license status and for other systems, such as payment providers and CRMs, to update license information (for example, during license extension).

of reasons why the ability to find outliers is useful. First, it can help vendors to detect fraudulent behaviour (for example, user trying different license key string combinations). This may be more applicable to the B2C case. However, since most users have good intentions (especially B2B cases), finding outliers can help vendors to detect when a customer might be experiencing problems and reach out to them. Moreover, it can give insights on if there are any changes in the way the application is being used.

Currently, a majority of software vendors without a dedicated data scientist need to go through all the logs themselves in Excel, which is inefficient. If a model can be created that would narrow down the amount of data that needs to be analyzed, it would already solve a large problem.

1.2 Project goal

To the best of our knowledge, there is to date no research in the field of anomaly detection in software licensing data. Therefore, our goal is to, first and foremost, investigate whether it is possible to find anomalies in software licensing data with unsupervised methods. We will examine this by attempting to develop a classifier that can aid vendors in finding outlying behaviour in the

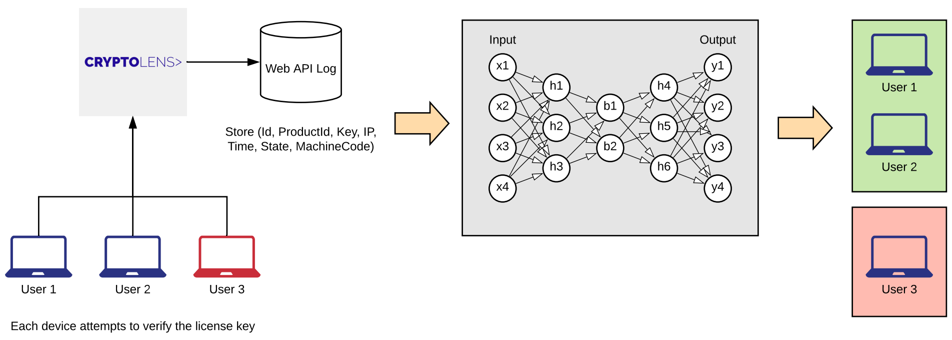
CRYPTOLENS>
Web API Log
Store (Id, ProductId, Key, IP, Time, State, MachineCode)
User 1
User 2
User 3
Each device attempts to verify the license key
Input
Output
x1
x2
x3
x4
h1
h2
h3
b1
b2
h4
h5
h6
y1
y2
y3
y4
User 1
User 2
User 3

Figure 1.2: The goal of the project consists of three steps: process the data, train an anomaly detection model and display which time-frames are anomalous.

large amount of data generated from license-related operations (e.g. license verifications) using latest unsupervised machine learning methods. The final model will be assessed based on its characteristics during training and inference phases. In the end, we want to produce a vendor-specific model that requires minimal maintenance. A summary of the process is shown in Figure 1.2.

1.3 Evaluation

The criteria below will be used as a guide when evaluating the models:

Training phase

- **Speed** – How quickly can a model be trained?
- **Stability** – How stable is the training? For example, does the training error explode at some point?
- **Performance** – How well does the model learn the underlying distribution of the data?

Inference phase

- **Speed** – How quickly can we infer the anomaly score or a binary label?

- **Performance on unseen data** – Does the model find anomalous samples correctly, i.e. is it worth for the vendor to take an extra look at them?
- **Interpretability** – How easily can the result be interpreted from a vendor's perspective? For example, the less the vendor needs to do the better. If we can give a binary label to a sample is better than a score.

Chapter 2

Background

In this chapter, we cover the relevant theory that will be used for anomaly detection.

2.1 Software licensing concepts

2.1.1 Terminology

- **License server** – License server allows customers to verify licenses. Each end user application instance will contact a license server to verify the status of a license or to synchronize changes.
- **B2B** – Refers to business-to-business sales, i.e. when a software vendor sells to a company that typically has 1000+ employees.[1]
- **B2C** – Refers to business-to-customer sales, when a software vendor sells to a consumers or SMEs.
- **License** – A license is an object that contains information about what an end user is permitted to do, for example, which features can be accessed, when the license will expire and on how many machines it can be used on. From an end user perspective, it is represented as a string, eg. `EENFJ-PXGWT-HZDIY-UKIPL`.
- **End user** – An end user is an actual user of the application (some systems refer to this as a *seat*). In the B2C case, a license typically belongs

[1]Note: from a licensing perspective, the number of employees should not be seen as a hard bound; companies with fewer employees may still have the same licensing needs as larger enterprises.

to one physical person and the end users are the devices that belong to that person. In the B2B case, when one license is typically given to the entire company, an end user is usually a specific employee. In Cryptolens, a license key can have a set of *machine codes* assigned to it, which can be seen as an end user. Note, the way an end user is defined is up to the software vendor. All examples use the device fingerprint as the definition of the end user (which means end users are the devices), but the software vendor is free to set this to something else. For example, in some scenarios, it can be set to a network name, device fingerprint combined with the processor id (in case multiple instances running on the same machine should be seen as separate end users) or a random string (when there is no easy way to identify an instance, especially in the case of docker containers, virtual environments or applications that keep turning on and off, such as lambda functions).

- **Vendor** – The software vendor is the one who will monetize the software application.
- **Customers** – The customers that will use the application that the software vendor has developed.
- **Node-locked license** – a node-locked license is when there is a limit of how many end users (in Cryptolens, it is the limit on the number of machine codes) that can be associated with a license. For example, if the vendor wants to restrict the maximum number of devices that can use the license key, node-locking can be used. In order to free up unused end users (for example, when employees leave), they need to be deactivated. The deactivation of an end user requires a separate method call.
- **Floating licensing** – a floating license is similar to node-locked licenses in the sense that it sets a limit for the number of end users, with the difference that unused end users are freed up automatically after a period of inactivity. The end user application instance, in contrary to node-locked licenses, needs to send periodic requests (aka. heartbeats) to the license server to remain active. If it fails to do so, it will automatically be deactivated and other end users will be able to use the license. As a result, customers can have the application installed on any number of computers but be limited to a certain number of concurrent end users. The time-window within which a application instance needs to send a request can be defined by the vendor.

- **Usage-based licensing** – The usage-based model allows vendors to charge customers based on an accumulated usage. For example, in the case of an accounting software, vendors could charge extra for generation of yearly reports. Customers could either get a fixed number of credits (i.e. on a pre-paid basis) or be charged for the actual number of credits used (i.e. on a post-pay basis) [1].

2.1.2 Amount of data per licensing model

Depending on the licensing model used, there will be different amount of data logged, which can affect how well a model can learn the data. The least amount of data will be for licenses using the node-locked model. Some customers will verify the license each time the application starts whereas others will verify it periodically, for example, once a year. Both usage-based and floating licenses will have more data since the user needs to be online to synchronize changes with the license server. The floating license model will generate most of the data.

2.2 Anomaly detection in time-series

This section is based on the recent survey from Cook, Misirli, and Fan [2] and describes the foundation needed to understand anomaly detection in time-series.

2.2.1 What is an anomaly?

Before we go in depth on how to find anomalies in a data set, we need to define what an anomaly is. A couple of definitions have been suggested. Cox [3] define it as "*an observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism*". In more general terms, an anomaly is "*the measurable consequences of an unexpected change in state of a system which is outside of its local or global norm*" [2]. In this study, we will assume that most of the data is normal and that anomalous data only constitutes a small fraction of it.

2.2.2 Types of anomalies

There are three types of anomalies: point anomalies, contextual anomalies and collective/pattern anomalies.

Point anomaly

A *point anomaly* is when a point suddenly goes beyond the normal range of values and later returns back to normal, i.e. if a time series contains the values $\{0.5, 0.3, 0.7, 0.1, 100, 0.7, 0.9\}$, the value 100 is a point anomaly since the normal state are values between zero and one.

Contextual anomaly

A *contextual anomaly* is when a collection of points or a sequence deviates from the expected pattern. On its own, the collection of points may still conform to the normal range of values, but when put into the context of other data points, it is clearly an anomaly. For example, if the normal state follows the sine curve, then if we suddenly get a point that is not on the sine curve, this would be a contextual anomaly, although on its own, the point would not be a point anomaly if it is within the range of a sine curve.

Collective/pattern anomaly

A *collective anomaly* is when a collection of points differs significantly from the rest of the data points. On their own, the points may not constitute an anomaly, only when they are viewed as a group. For example, we have the sequence $\{0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1\}$. If we split it into groups of 3 each, we get $\{0, 0, 0\}$, $\{0, 1, 0\}$, $\{1, 1, 1\}$, $\{0, 0, 1\}$, it can be clearly seen that the 3rd groups differs significantly from the other groups since it consist of only ones. If we would view the points in isolation, we would not have detect this outlying behaviour.

2.2.3 How to classify an anomaly

Depending on the method, there are different way to classify anomalies: either using an anomaly score or binary labels.

Anomaly score

An anomaly score is a values that specifies to what extent a point (or a group of points) constitute an anomaly. The way this value is computed depends on the method. For example, if an autoencoder is used, then the reconstruction error could serve as a basis for the anomaly score.

Binary labels

Some methods can assign a binary label to a point (i.e. if it is an anomaly or not). For example, if a clustering method such as KMeans or Gaussian Mixture Models are used, they can provide us with a label automatically. In other cases, a threshold needs to be introduced, which is then used to determine the labels. For example, in the case of an autoencoder, we can set a threshold θ on the reconstruction loss, so that points above θ are classified as anomalies.

2.3 Methods to detect anomalies

A common method to detect anomalies is to use an autoencoder network. There are two approaches an autoencoder network can be utilized, either by only using the autoencoder or combining it with a clustering method. In the first case, the reconstruction error is used as the basis for an anomaly score. An existing problem with this approach is a way to decide the threshold. In the second case, only the encoder part is used to reduce the dimension of the data, so that this new representation can be analyzed by a clustering method. In both cases, the goal is to first train an autoencoder that attempts to learn the "normal" distribution of the data set.

2.3.1 Autoencoder for anomaly detection

An autoencoder is a neural network whose goal is to reconstruct the input with minimal loss metric. It can be thought of as an identity function, i.e. we want to train a network so that $f(x) \approx x$. The network is designed so that each

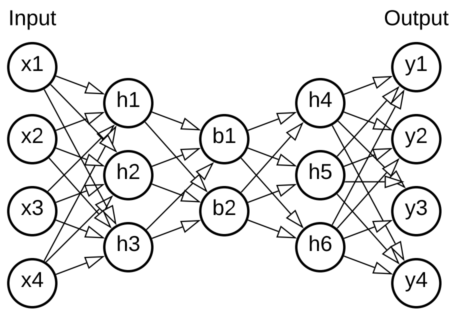

Figure 2.1: An example of an autoencoder that takes in an input vector of size 4 and attempts to compress it into a vector of size 2. The bottleneck layer contains the nodes $b1$ and $b2$.

subsequent hidden layer has lower dimension than the input until a bottleneck layer is reached, after which the number of nodes is increased again until it is of the same dimension as the original input. The part of the network from the input vector to the bottleneck layer is referred to as the **encoder** and the **decoder** is the network from the bottleneck to the output.

The bottleneck forces the autoencoder to find a compressed representation of the input. In fact, the compressed representation (i.e. the output of the encoder) was proven to be related to principle component analysis in [4]. Thus, an autoencoder can be viewed as a dimensionality reduction method. An example of an autoencoder network is shown in Figure 2.1, where the network attempts to compress the input of size 4 into a vector of size 2. If the network is able to find an optimal representation, the reconstruction error will be small.

Early attempts

An early attempt to tackle anomaly detection in an unsupervised manner using autoencoders was described in [5], where Replicator Neural Network (RepNN) was proposed. The RepNN in [5] is an autoencoder that has three hidden layers that are fully connected, whose goal is to learn to reconstruct each sample with a small mean square error (MSE). The number of neurons in each hidden layer is decreased until we reach the bottleneck layer, after which the number of neurons in the hidden layers is increased. The last layer contains the same number of nodes as the input layer (see Figure 2.1). Thanks to the bottleneck layer, the network is forced to find a compressed representation of the data. As a result, normal data will have a small MSE whereas outliers will have a large MSE. An anomaly is defined to be a sample that has an error greater than a certain threshold. In order to improve the performance of a RepNN, [6] has suggested to use dropout layers and [7] to divide training into several stages where data is first split into two sets, normal and anomalous, in an unsupervised manner, and later train another model only on the normal data in order to avoid underfitting.

2.3.2 Using LSTM/TCN based autoencoder

We can build on the idea with autoencoders for anomaly detection so that it is better suited for time-series. Recurrent neural networks (RNNs) offer a way to capture relationships in data over a period of time [2]. Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) are preferred over vanilla RNNs that suffer from the vanishing gradient problem [2]. When LSTMs are used in an autoencoder, it has been shown to be effective at finding anomalies in both univariate and multivariate time-series [8, 9]. In [9], it is shown that anomalies can be found in multivariate time series with the length between 30-500. To achieve better performance, several LSTMs can be stacked together so that more temporal information can be learned [8, 10].

Recently, temporal convolutional neural networks (TCNs) have been suggested as an improvement over RNN based methods [11, 12]. TCNs are both faster to train and have been shown in [12] to outperform canonical LSTMs in a wide range of tasks. A TCN, as Bai, Kolter, and Koltun [12] put it is

> TCN = 1D Fully-convolutional network (FCN) + causal convolutions

The use of 1D fully convolutional network ensures the size of the output is the same as the input and the causal convolutions ensure that there is no leakage from the future to the past [12]. In [13], there is an example how TCN based autoencoder can be used in anomaly detection.

2.3.3 Finding the classification threshold

A problem with unsupervised anomaly detection using purely the autoencoder approach is finding a suitable threshold. In most studies where this method was used, it is assumed that we either have labels or that a certain percentage of the data is anomalous. Fernández-Saúco et al. [14] suggests an automated method that has been shown to find a threshold close to the correct one on two datasets (see Appendix A for the pseudo code implementation). In [6], the threshold is based on the largest reconstruction error on the training set when outliers are removed. However, there does not seem to be a general way of determining the threshold. Thus, it may help to use an autoencoder in combination with another method, as described in the next section.

2.3.4 Encoder together with a clustering method

Since autoencoders reduce the dimensionality of the data in the bottleneck layer, we can use the encoder as a feature extractor and combine it with a clustering method, such as k-means, T-SNE [15] or Gaussian mixture models (GMMs). A description of how they work is available in the next chapter. For example, in [16], they first train a TCN so that it can be used to extract features in a lower dimension and then train a GMM based on the new representation. If an autoencoder is able to learn to reproduce normal samples correctly, we anticipate that its representation in the lower dimension will differ significantly from the corresponding representation of anomalous samples, as was shown in [17].

2.4 Clustering methods

Clustering methods offer a way to group data points in an unsupervised manner. Since they can be combined with the autoencoder and provide us with a more convinient way of classifying anomalies, a couple of methods will be described.

2.4.1 k-means

k-means is a distance-based clustering method that aims to minimize *within-cluster sum-of-squares* [18], i.e.

$$\sum_{i=0}^{n} \min_{\mu_j \in C}(||x_i - \mu_j||^2) \tag{2.1}$$

where $X = \{x_1 \dots x_i\}$ are the data points and $C = \{\mu_1 \dots \mu_i\}$ are the cluster means. The method works by first initializing k clusters, preferably far away from each other, and then two steps are performed in a loop. First, data points are assigned to the nearest cluster using Euclidean distance. After that, the centroid of each cluster is re-computed as a mean of the points in the cluster. This procedure can be repeated until the cluster means remain stable. Convergence is guaranteed, but it may not be optimal.[18]

2.4.2 Gaussian Mixture Models

Gaussian mixture model is a probabilistic method that assumes that "*all points are generated from a finite number of Gaussian distributions with unknown parameters*" [19]. It can be thought of as a generalization of the k-means method, since it also takes into account the covariance structure of the data [19]. It uses the Expectation-Minimization method to find which point belongs to which cluster.

Chapter 3

Methods

In this section we will go through the data processing pipeline (Section 3.1), the privacy measures taken throughout the project (Section 3.2), methods used to detect anomalies (in Section 3.3) and the evaluation metrics to assess the quality of the anomalies (in Section 3.4).

3.1 Data

In the following section the data processing pipeline is described. A summary of the steps is shown in Figure 3.1. In sum the steps are as follows. First, the raw data for a specific user is retrieved. Secondly, most columns are normalized and additional columns are added that can be derived from the raw data (for example, the country from the IP address). Thirdly, the data is split up into chunks of 100 requests each. After that, the time column is normalized and the chunks are split into a training and test sets with the 80:20 ratio.

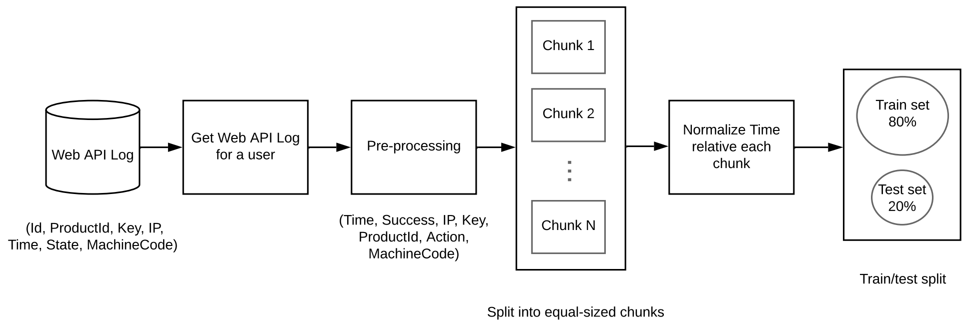
Web API Log
(Id, ProductId, Key, IP, Time, State, MachineCode)
Get Web API Log for a user
Pre-processing
(Time, Success, IP, Key, ProductId, Action, MachineCode)
Chunk 1
Chunk 2
Chunk N
Split into equal-sized chunks
Normalize Time relative each chunk
Train set 80%
Test set 20%
Train/test split

Figure 3.1: Overview of the data preparation pipeline.

3.1.1 Data source

The source of the data to be analyzed comes from the Web API Log [20]. Each request that is related to a license key is logged, for example license key activation, deactivation, change of a dataobject associated with a license, etc. The structure of each log entry is shown in Figure 3.3. The way the raw data is gathered is shown in Figure 3.2.

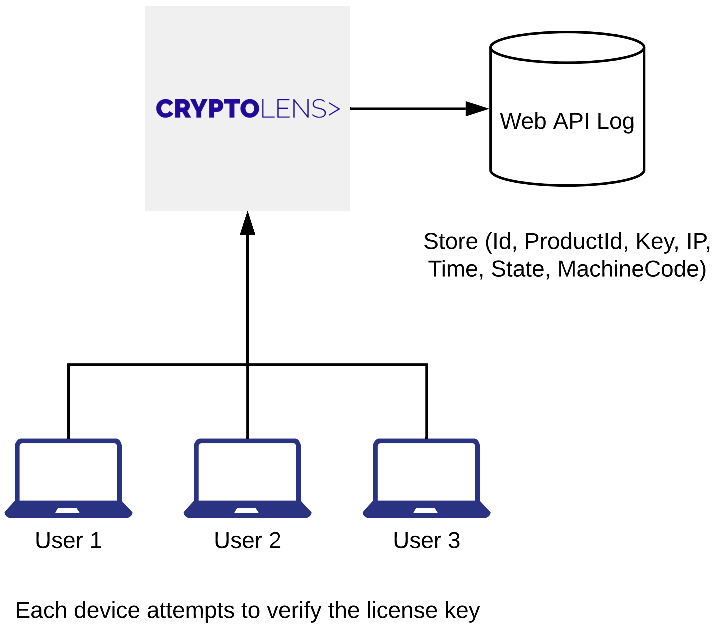
CRYPTOLENS>
Web API Log
Store (Id, ProductId, Key, IP, Time, State, MachineCode)
User 1
User 2
User 3
Each device attempts to verify the license key

Figure 3.2: Explains how data is gathered in the Web API Log. When a device attempts to verify the license, Cryptolens stores the result in the Web API Log.

3.1.2 Pre-processing

Before the data can be used to train the models, it needs to be preprocessed. This is done for three reasons. First, LSTMs and TCNs require a numerical value as input and preferably normalized. Secondly, the **state** column contains compressed information about the status of the request, which have different degrees of importance. Thus, it may be better to split it into separate columns so that the model can determine their importance. Finally, the IP address can provide us with geographical information, which can be useful for the model to know. For example, even if IP addresses change frequently, it may not be a

Id	A unique identifier of the event.
ProductId	The id of the product.
Key	The license key string.
IP	An anonymized IP of the device that triggered this log.
Time	The time when this log was created.
State	Contains more information about the method that was called and if it was successful or not.
MachineCode	The machine code of the device, if applicable.

Figure 3.3: The information stored inside the Web API Log.

root of concern assuming that the country is the same. However, if the country changes as well, we want the model to take that into account too.

As a result, we will add three new columns: **success** (whether the request was successful or not), the **action** (a number specifying what action was invoked) and **country** (a numerical representation of the country). Since the **Id** does not contribute any useful information about the state of an action, it will be disregarded.

The first step in the pre-processing pipeline of user data is to ensure that it only contains normalized numerical value. In order to convert all values to an integer, each value, no matter if it is a string or an integer, is mapped to an index inside a dictionary. This is performed for all data since the creation of the account. In other words, if a user has only activated three devices with machine codes *AAA*, *BBB* and *CCC*, they would be mapped to values 0, 1 and 2, respectively. Once the data is in numerical form, we normalize it by dividing by the size of the dictionary (i.e. the number of distinct values in a column). Going back to the three machine codes, they would be represented as 0.33, 0.66 and 1, respectively. Since the time column will always contain different values for each row, we do not normalize it in this step. The list of all columns used in the analysis is illustrated in Figure 3.4.

The next step is to split the data into equally-sized chunks. Our choice was to split it into chunks of size 100. The splitting phase can be accomplished in two ways: with or without overlap. Once we have split the data into chunks, the **time** column is normalized relative to each chunk. In other words, it will always start at zero and end with 1 in each chunk.

Finally, once the time-series is split into equally-sized chunks, they are shuffled and split into a training and test set with the ratio of $80 : 20$.

Time	The time relative to each chunk.
Success	Whether the request was successful (0 or 1).
IP	An anonymized IP in normalized form.
Key	Normalized license key.
ProductId	Normalized id the of the product.
Action	Normalized integer representing details of the request.
MachineCode	Normalized machine code.
Country	Normalized country derived from the IP.

Figure 3.4: The information that will be analyzed by the model. Note, all columns but time are normalized relative to all values observed since the creation of the account.

3.1.3 Test cases

Models will be evaluated on three cases so that the performance can be assessed for different types of licensing models. A summary is shown in Figure 3.5.

Case	Licensing model	Training size	Test size
1	Standard (node-locked)	32	8
2	Usage-based licensing (mostly)	80	20
3	Floating licensing	19004	4751

Figure 3.5: The different cases that will be tested. Each case represents a customer, which is why it may contain requests using other licensing model. Each sample is a set of 100 consecutive requests.

3.2 Privacy measures

In order to safeguard privacy, only the information necessary to perform the experiment was used. All IP addresses were anonymized, notes field and customer ids were removed. During testing, IP addresses, product and key ids were scrambled further so that they did not convey any more information than that of a change of the content in a field. Moreover, there was no way to link a certain case to a specific customer. Analysis was performed on a computer in Sweden and all files were removed after the experiment.

3.3 Models

Anomalies can be found in two ways: either using the reconstruction error in the autoencoder or by using the latent representation of a trained autoencoder in combination with a clustering method. When using the first approach, a method is needed to set the threshold, which is described in a separate section.

3.3.1 Autoencoder-based models

Two different types of autoencoders will be evaluated, based on LSTM and TCN layers. In both cases, we will test different number of hidden layers and the size of the bottleneck layer. The TCN autoencoder implementation is based on the one in [13] and the LSTM autoenconder is based on the implementation from [21]. The exact definition of the networks used is documented in Appendix B.

3.3.2 Finding the threshold

We will use three methods to find the classification threshold for anomalies given the reconstruction error. The first method uses the algorithm suggested in [14]. The second method uses the largest reconstruction error observed on the training set, which was used in [6]. The third method is to use the largest error obtained using a histogram of errors. Implementation of these methods is available in Appendix A.

3.3.3 Autoencoder with clustering

When an autoencoder has been trained, a clustering method will be trained the compressed representation of the test data. To obtain the compressed representation, the encoder network will be used. KMeans and GMM will be used as the clustering method. We will use the `sklearn`[1] implementation of KMeans and GMM. Given that we have extracted the encoder in `enc`, they can be called as shown below:

```python
from sklearn.mixture import GaussianMixture
from sklearn.cluster import KMeans

kmeans= KMeans(n_clusters=2)
pred = kmeans.fit_predict(enc.predict(test))

```

[1]More information can be found here: `https://scikit-learn.org/stable/`

```
gmm = GaussianMixture(n_components=2)
pred2 = gmm.fit_predict(enc.predict(test))
```

3.4 Assessing performance

Since there is no objective metric to assess the performance of a model (because there are no "*correct*" labels), it needs to be performed manually. To aid in this comparison, we need to first summarize the data and compare the summary of a normal sample with the one the model classifies as anomalous.

To summarize the data, we can compute the number of unique items in each column, which will result in a 1-dimensional vector. For example, given three log items with three columns each, $\{(0, 1, 1), (0, 2, 1), (0, 3, 2)\}$, the summary vector would be $(1, 3, 2)$. We can then compare the difference between the number of unique items for normal and anomalous samples.

Note, in this analysis, the distribution of time will not be taken into account and is solely based on the method described above.

Chapter 4

Results & Discussion

Four types of results were collected to evaluate the models. The quality of the samples that were classed as anomalies are summarized in Figures 4.1, 4.2 and 4.3. The distribution of the errors is shown in Figure 4.4. The speed of each model is illustrated in Figure 4.5. The evolution of the training and validation loss across the epochs is available in Figures 4.6, 4.7 and 4.8.

The *model name* refers to the architecture of the network, which are described in more detail in Appendix B. The *bottleneck* is the shape of the last layer in the encoder before the size of subsequent layers starts to increase. The *depth* is defined in two ways, depending on if it is an LSTM or a TCN network. For an LSTM network, it is the number of hidden LSTM layers and for the TCN network it is the the number of convolutional layers. Using each of the networks, five experiments were performed. Method 1 (M1), Method 2 (M2) and Method 3 (M3) refer to the different methods to find a decision threshold, so the result is the number of anomalies given the threshold. In the last two columns, GMM and KMeans is when the autoencoder network is first trained and later the encoder is used in combination with a clustering method. GMM and KMeans will attempt to find two classes and we will treat the class with fewest members as the anomaly class.

In Figures 4.1, 4.2 and 4.3, different colours are used as a way to convey the quality of the found anomalies, using the method in Section 3.4. Green values mean the samples were clearly an anomaly, yellow is when it is unclear or on the border line and red is when it was clearly not an anomaly.

The rest of the results section will focus on evaluating the decision threshold method (Section 4.1), whether it is better to use the reconstruction error of an autoencoder or let a clustering method take the decision (Section 4.2), which model architecture performs better (Section 4.3), which model is better given the constraints (Section 4.4), ethical implications and sustainability (Section 4.5) and future work (Section 4.6).

Model name	Bottleneck	Depth	M1	M2	M3	GMM	KMeans
LSTM_AE	(16,)	1	7	0	0	3	3
LSTM_AE_Small	(3,)	1	na	0	0	4	1
LSTM_AE_Deep	(8,)	2	6	1	1	0	1
LSTM_AE_Deep2	(4,)	3	na	1	1	2	2

Model name	Bottleneck	Depth	M1	M2	M3	GMM	KMeans
TCN_AE	(12,6)	4	na	0	0	8	1
TCN_AE2	(3,)	4	na	0	0	0	4
TCN_AE_Small	(6,3)	5	na	0	0	0	0
TCN_AE_Small2	(3,)	5	na	0	0	0	4

Figure 4.1: The number of anomalies detected in case 1 (node-locking licensing). The test size is 8 (see Figure 3.5). Green values mean it is clearly an anomaly, yellow is when it is unclear or on the border line and red is when it was clearly not an anomaly. "M1", "M2" and "M3" refer to methods to find the threshold, which are described more in detail in Appendix A. "na" is when the method was unable to find a threshold.

4.1 Decision threshold method

Based on Figures 4.1, 4.2 and 4.3, out of the three methods that were tested to decide the decision threshold, method 1 performed the worst. It took long time for the method to converge and once it did the result gave mostly false positives. Method 2 was restrictive in what was classified as anomaly (since few anomalies were found, as can be seen in Figures 4.1, 4.2 and 4.3), but the quality of the result was good (since most of the anomalies had green value). Method 3 found more anomalies than method 2 and the quality was equally good. As a result, we would recommend to use method 3 as a way to find the decision threshold since it is fast and gives high quality anomalies. Moreover, method 3 can be tuned by changing which error is used from the histogram. An example of an error histogram is available in Figure 4.4.

Model name	Bottleneck	Depth	M1	M2	M3	GMM	KMeans
LSTM_AE	(16,)	1	14	1	1	9	9
LSTM_AE_Small	(3,)	1	20	0	1	2	7
LSTM_AE_Deep	(8,)	2	na	0	0	5	5
LSTM_AE_Deep2	(4,)	3	na	0	1	0	1

Model name	Bottleneck	Depth	M1	M2	M3	GMM	KMeans
TCN_AE	(12,6)	4	na	0	0	0	0
TCN_AE2	(3,)	4	na	0	0	0	0
TCN_AE_Small	(6,3)	5	na	0	0	0	0
TCN_AE_Small2	(3,)	5	na	0	1	0	0

Figure 4.2: The number of anomalies detected in case 2 (usage-based licensing). The test size is 20 (see Figure 3.5). Green values mean it is clearly an anomaly, yellow is when it is unclear or on the border line and red is when it was clearly not an anomaly. "M1", "M2" and "M3" refer to methods to find the threshold are described more in detail in Appendix A. "na" is when the method was unable to find a threshold.

4.2 Autoencoder with or without clustering

When deciding whether to use the reconstruction error of an autoencoder or combine the encoder network with a clustering method, it appears, based on the results in Figures 4.1, 4.2 and 4.3, that the first approach using the reconstruction error is better. There are three reasons for that. First, the clustering methods are inflexible since their tolerance cannot be adjusted. Secondly, what clustering methods classified as an anomaly was either wrong or on the border line (because the anomalies had either red or yellow value). Finally, for shallower networks, clustering methods performed worse than when using a method based on the reconstruction error. As a result, our recommendation is to use the approach based on the reconstruction error for all types of networks. Encoder network with a clustering network may work but requires deeper autoencoder network.

4.3 LSTM vs. TCN

When comparing LSTM vs TCN based autoencoder, it seems that the LSTM based autoencoder was able to find more anomalies that were of good quality in comparison to the TCN based autoencoder (which still found anomalies, but not all of them). This can be seen by comparing the number of green values in

Model name	Bottleneck	Depth	M1	M2	M3	GMM	KMeans
LSTM_AE	(16,)	1	na	0	0	1131	2361
LSTM_AE_Small	(3,)	1	na	0	1	213	2
LSTM_AE_Deep	(8,)	2	na	0	2	0	0
LSTM_AE_Deep2	(4,)	3	na	0	5	1	1
Model name	Bottleneck	Depth	M1	M2	M3	GMM	KMeans
TCN_AE	(12,6)	4	na	0	2	0	0
TCN_AE2	(3,)	4	na	0	2	0	3
TCN_AE_Small	(6,3)	5	na	1	2	0	0
TCN_AE_Small2	(3,)	5	na	1	2	0	3

Figure 4.3: The number of anomalies detected in case 3 (floating licensing). The test size is 4751 (see Figure 3.5). Green values mean it is clearly an anomaly, yellow is when it is unclear or on the border line and red is when it was clearly not an anomaly. "M1", "M2" and "M3" refer to methods to find the threshold are described more in detail in Appendix A. "na" is when the method was unable to find a threshold.

column *M3* in Figures 4.1, 4.2 and 4.3, since more green value anomalies were found with the LSTM based models. In Figures 4.1 and 4.2 that were trained on a small dataset, the TCN is restrictive in what it classifies as anomaly. Only the LSTM based methods seem to find most of the anomalies. When the dataset is large (as in Figure 4.3 does the TCN find more anomalies (but not as many as the LSTM based models). For both LSTM and TCN networks, it appears that the deeper the network, the more quality anomalies are found. The network depth also affects the performance of the clustering methods (the deeper the network, the better the clustering method performs), if the encoder is used as a dimensionality reduction tool. The amount of data in the training set does seem to affect the performance too. In Figure 4.4, when the training set is large (as in case 3, floating licensing) most normal samples have the same error whereas anomalies are the small number of samples with a large error. In other cases where the training set is small, it is not as clear what should be regarded as an anomaly.

When comparing the speed of LSTM and TCN based autoencoders, TCN based autoencoders are clearly faster (by a factor of 5-10), as can be seen in Figure 4.5. As the dataset increases in size, the training time of the deepest LSTM network is 10 times slower than the deepest TCN network.

The convergence of models, shown in Figures 4.6, 4.7 and 4.8, seems to occur faster for LSTM based methods when the dataset is small. On the contrary, when the amount data is large, there does not seem to be any difference in the time to converge. A reason to prefer a TCN based model is because the loss curve is smoother. With LSTM based models, the loss curve has sudden spikes. In some cases, we had to restart training because the error would suddenly increase and not improve over time. For smaller datasets, the LSTM seems to be better at learning the data.

4.4 Recommended model

The goal of the project was to find the best model for anomaly detection given the criteria defined in Section 1.3. In the training phase, the criteria were the speed, stability and performance and in the test phase it was the inference speed, performance on unseen data and interpretability.

If the training speed is important, a TCN based autoencoder is a better choice. It is more picky on what it considers an anomaly in comparison to LSTM based autoencoder, so it may not find all anomalies, but those that it finds are clearly anomalies. TCN models are also more stable to train. They do not have spikes in the loss curve as the LSTM models and the evolution of the loss over the epochs is smooth. It may take longer to converge for a TCN model but, since they are fast to train, this is not a problem.

In the inference phase, the speed was affected by whether an LSTM or TCN based autoencoder was used. When comparing the performance of different methods to find the threshold vs. using a clustering method, we found method 3 (the histogram method) on the reconstruction error to be better. Combining the autoencoder with a clustering method is better for interpretability, but since the quality of the anomalies they found was not as good, we do not recommend it. More research is needed to determine if clustering methods can be a good alternative. Instead, we recommend to either base the decision on the largest error in the histogram or present the user the reconstruction error and the error histogram so that they can analyse it on their own.

4.5 Ethics and sustainability

The project was executed with privacy in mind, making sure that no end user can be determined and ensuring that there is no way to link the data to a certain customer. The results can have both positive and negative ethical implications. A downside is that using the created models, there will be less work for those who normally analyzed the data in house. On the positive side, the results from the model will allow companies to be more efficient and empower the SMEs who would not normally be able to afford in house data analyst or outsource it.

4.6 Future work

Recently, a paper was published about an improved version of an LSTM, *mogrifier LSTM* (see [22]), which we think could be interesting to evaluate on licensing data. Moreover, we suggest to test if *Generative Adversarial Networks* could be an alternative to the autoencoder network. Furthermore, a couple of additional experiments can be performed with the existing models, for example, use different chunk sizes (we used 100 log entries per chunk) as well as allow overlapping between chunks (which can be helpful when the amount of user data is small).

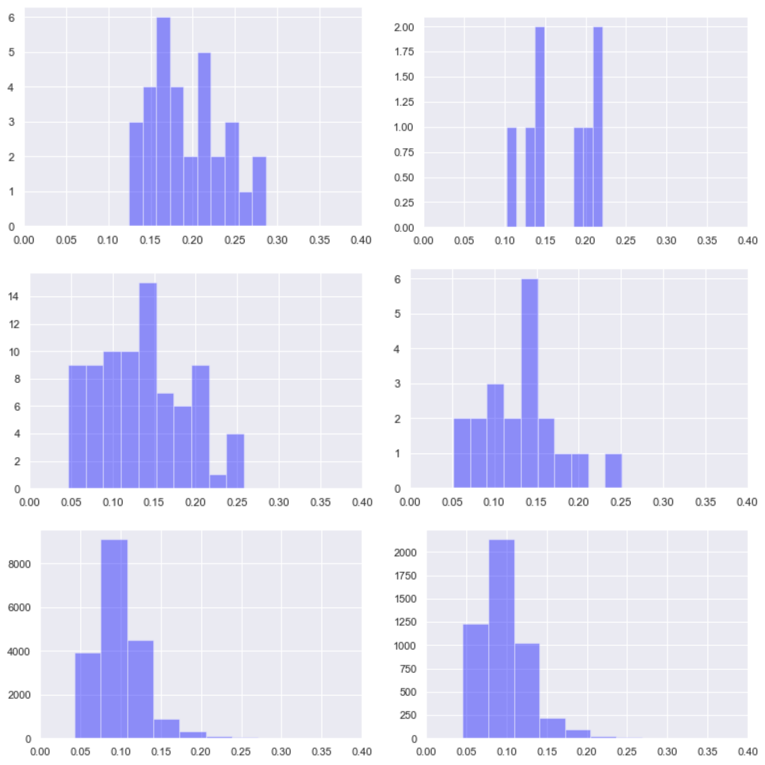

Figure 4.4: Histograms of the errors for the three data sets using LSTM_AE_Deep2 (error on the x-axis and the number of samples on the y-axis). The left graphs are the training error distribution and the right ones are the test error distribution. The top graphs are case 1, the middle ones are case 2 and the bottom ones are case 3, as defined in Figure 3.5. As can be seen in the bottom row, when the model was able to learn the normal cases, most of the errors are the same and only a few samples have a large error, which is what we consider an anomaly. With smaller training set size, it is not as clear what an anomaly is.

Model name	Dataset	Bottleneck	Depth	Time (10 epochs) [s]
LSTM_AE	standard (case 1)	(16,)	1	16
LSTM_AE_Small	standard (case 1)	(3,)	1	7.9
LSTM_AE_Deep	standard (case 1)	(8,)	2	12.7
LSTM_AE_Deep2	standard (case 1)	(4,)	3	15.7
TCN_AE	standard (case 1)	(3,)	4	2.8
TCN_AE2	standard (case 1)	(3,)	4	2.7
TCN_AE_Small	standard (case 1)	(6,3)	5	3
TCN_AE_Small2	standard (case 1)	(3,)	5	3.8
LSTM_AE	usage-based (case 2)	(16,)	1	12.7
LSTM_AE_Small	usage-based (case 2)	(3,)	1	11.1
LSTM_AE_Deep	usage-based (case 2)	(8,)	2	16.2
LSTM_AE_Deep2	usage-based (case 2)	(4,)	3	21.6
TCN_AE	usage-based (case 2)	(3,)	4	3
TCN_AE2	usage-based (case 2)	(3,)	4	3.1
TCN_AE_Small	usage-based (case 2)	(6,3)	5	3.8
TCN_AE_Small2	usage-based (case 2)	(3,)	5	3.8
LSTM_AE	floating (case 3)	(16,)	1	1067.8
LSTM_AE_Small	floating (case 3)	(3,)	1	938.1
LSTM_AE_Deep	floating (case 3)	(8,)	2	1627.3
LSTM_AE_Deep2	floating (case 3)	(4,)	3	2000.5
TCN_AE	floating (case 3)	(3,)	4	214.4
TCN_AE2	floating (case 3)	(3,)	4	216.2
TCN_AE_Small	floating (case 3)	(6,3)	5	202.8
TCN_AE_Small2	floating (case 3)	(3,)	5	217.8

Figure 4.5: Summary of the time it took to train the model for 10 epochs.

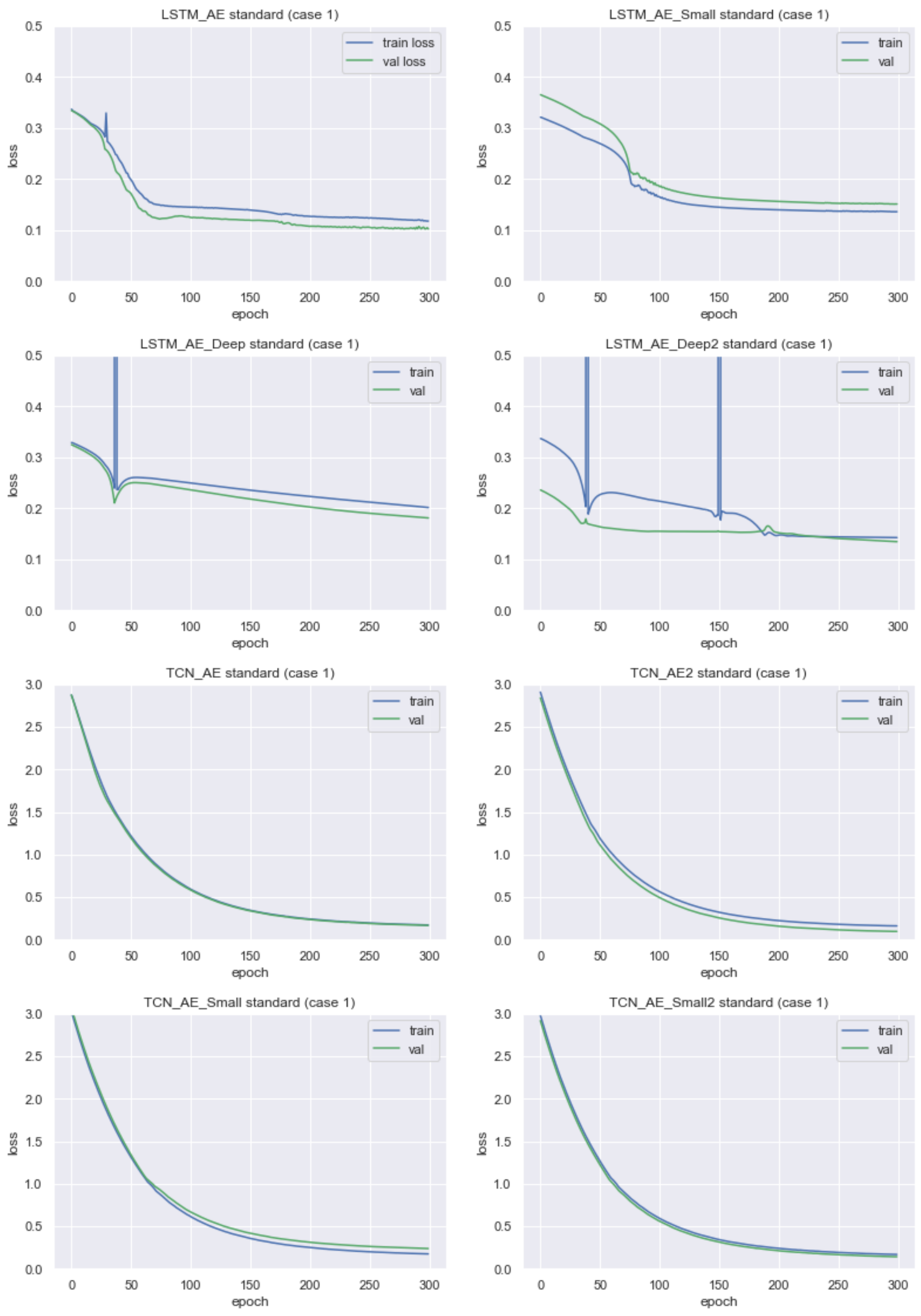

Figure 4.6: The evolution of the training and validation loss across the epochs for the standard (case 1) dataset (as defined in Figure 3.5). The loss metric is MAE.

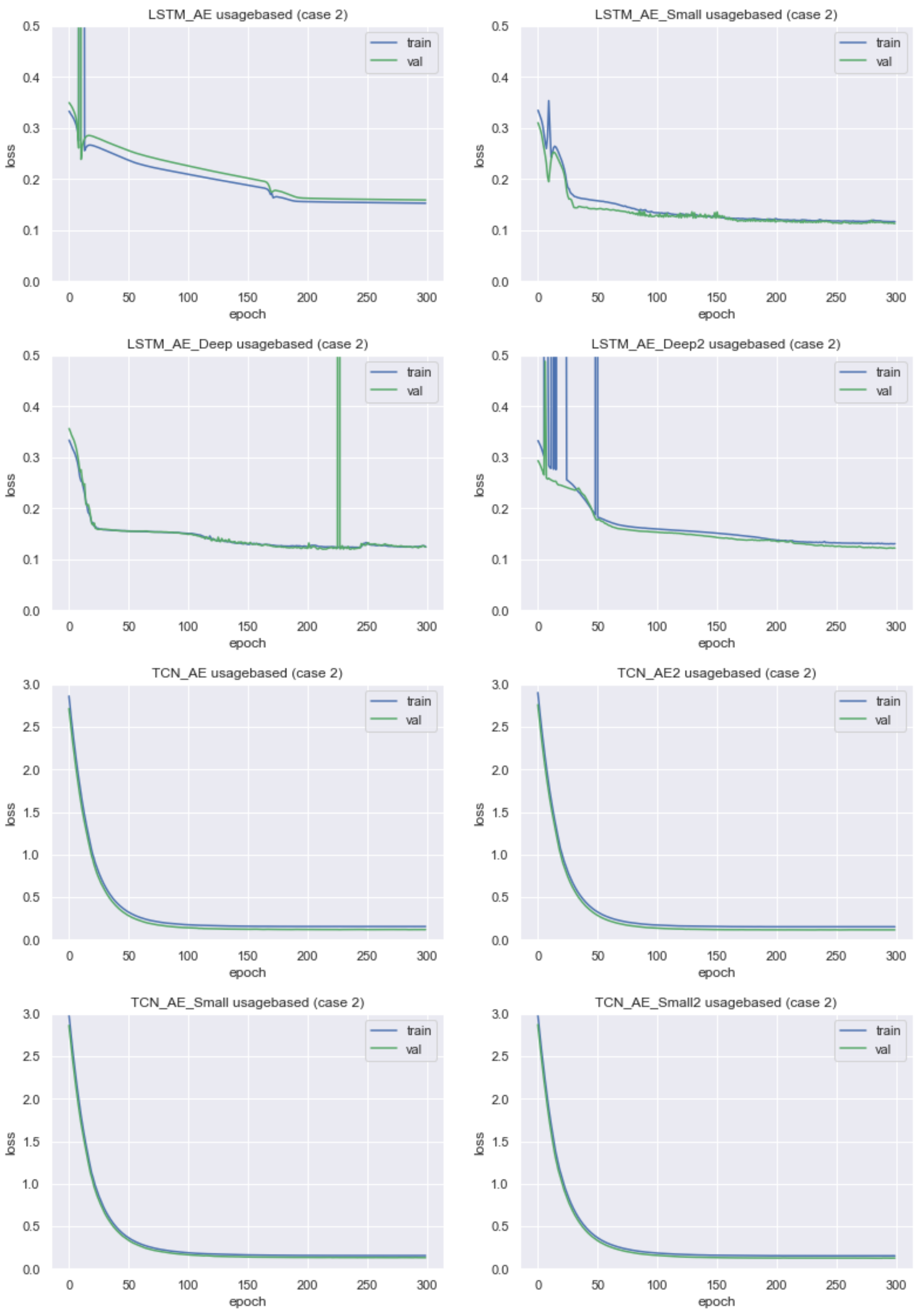

Figure 4.7: The evolution of the training and validation loss across the epochs for the usage-based (case 2) dataset (as defined in Figure 3.5). The loss metric is MAE.

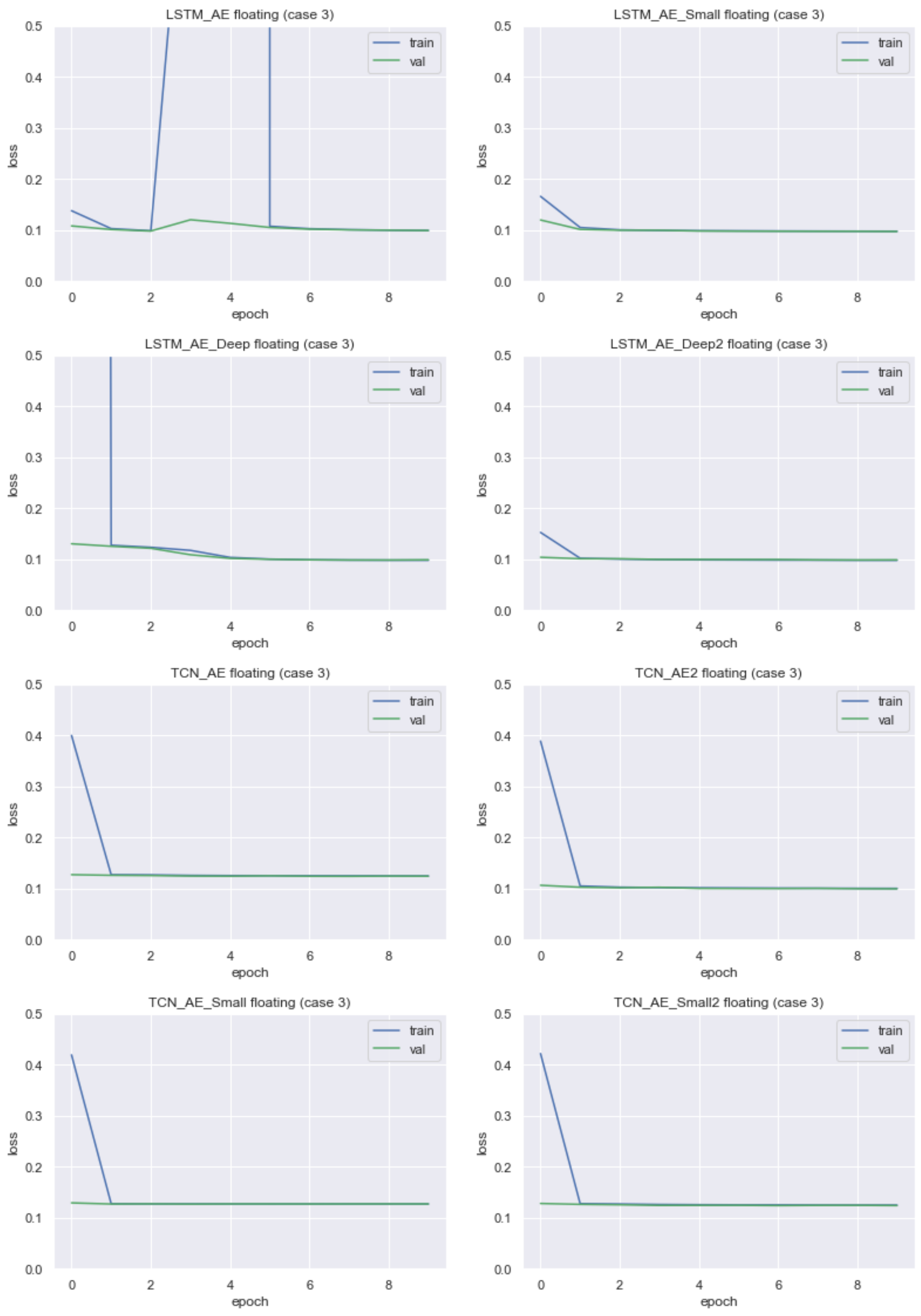

Figure 4.8: The evolution of the training and validation loss across the epochs for the floating (case 3) dataset (as defined in Figure 3.5). The loss metric is MAE.

Chapter 5

Conclusions

In this thesis, we tested different unsupervised methods based on an autoencoder architecture to detect anomalies in software licensing data. The reason such analysis is needed is to aid software vendors to find anomalous behaviour, which can be fraud or a user who has issues with the software. In both cases, the goal is to allow vendors to automatically detect such behaviour without having to analyse all of the data themselves. As a result, it can minimize loss of revenue due to fraud and contribute to a better customer experience.

In sum, we found that the best performing model is when the threshold of reconstruction error of an autoencoder is determined by the largest error in the histogram of errors on the train set (method 3). Both LSTM and TCN based models can be used, where LSTM models tend to work better on smaller datasets and also find more anomalies, whereas TCNs are more stable during training and also faster than LSTMs. We would recommend to have a dataset of at least 10k samples for better performance.

Bibliography

[1] *Usage-based licensing*. Mar. 9, 2020. URL: `https://help.cryptolens.io/licensing-models/usage-based`.

[2] Andrew Cook, Goksel Misirli, and Zhong Fan. "Anomaly Detection for IoT Time-Series Data: A Survey". In: *IEEE Internet of Things Journal* (2019), pp. 1–1. ISSN: 2327-4662.

[3] D. M Cox D. R.;Hawkins. *Identification of Outliers*. Dordrecht: Springer Netherlands, 1969. ISBN: 9789401539968.

[4] R Hecht-Nielsen. "Replicator neural networks for universal optimal source coding." In: *Science (New York, N.Y.)* 269.5232 (1995), pp. 1860–1863. ISSN: 0036-8075. URL: `http://search.proquest.com/docview/733233332/`.

[5] S. Hawkins et al. "Outlier detection using replicator neural networks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2454. 2002, pp. 170–180. ISBN: 3540441239.

[6] Carlos Garcia Cordero et al. "Analyzing flow-based anomaly intrusion detection using Replicator Neural Networks". In: *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2016, pp. 317–324. ISBN: 9781509043798.

[7] Gaku Kotani and Yuji Sekiya. "Unsupervised Scanning Behavior Detection Based on Distribution of Network Traffic Features Using Robust Autoencoders". In: *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. Vol. 2018-. IEEE, 2018, pp. 35–38. ISBN: 9781538692882.

[8] Pankaj Malhotra et al. "Long short term memory networks for anomaly detection in time series". In: *ESANN 2015 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges (Belgium)*. Vol. 89. 2015.

[9] Pankaj Malhotra et al. "LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection". In: *ICML 2016 Anomaly Detection Workshop*. 2016. arXiv: `1607.00148 [cs.AI]`.

[10] Michiel Hermans and Benjamin Schrauwen. "Training and Analysing Deep Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., 2013, pp. 190–198. URL: `http://papers.nips.cc/paper/5166-training-and-analysing-deep-recurrent-neural-networks.pdf`.

[11] C. Lea et al. "Temporal convolutional networks: A unified approach to action segmentation". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9915. Springer Verlag, 2016, pp. 47–54. ISBN: 9783319494081.

[12] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". In: *CoRR* abs/1803.01271 (2018). arXiv: `1803.01271`. URL: `http://arxiv.org/abs/1803.01271`.

[13] Sridhar Alla. *Beginning anomaly detection using Python-based deep learning : with Keras and PyTorch*. 1st ed. 2019.. 2019. ISBN: 1-4842-5177-6.

[14] I.A. Fernández-Saúco et al. "Computing anomaly score threshold with autoencoders pipeline". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11401. Springer Verlag, 2019, pp. 237–244. ISBN: 9783030134686.

[15] Tejas Khot. "Visualizing high-dimensional data". In: *XRDS: Crossroads, The ACM Magazine for Students* 23.2 (2016), pp. 66–67. ISSN: 15284972.

[16] J. Liu et al. "Anomaly detection for time series using temporal convolutional networks and Gaussian mixture model". In: *Journal of Physics: Conference Series*. Vol. 1187. 4. Institute of Physics Publishing, 2019.

[17] Mayu Sakurada and Takehisa Yairi. "Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction". In: *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. MLSDA'14. Gold Coast, Australia QLD, Australia: Association for Computing Machinery, 2014, pp. 4–11. ISBN: 9781450331593.

DOI: 10.1145/2689746.2689747. URL: https://doi.org/10.1145/2689746.2689747.

[18] *K-Means*. Feb. 19, 2020. URL: https://scikit-learn.org/stable/modules/clustering.html#k-means.

[19] *GMM*. Feb. 19, 2020. URL: https://scikit-learn.org/stable/modules/mixture.html#gmm.

[20] *Web API Log*. Jan. 26, 2020. URL: https://app.cryptolens.io/docs/api/v3/model/WebAPILog.

[21] *LSTM Autoencoder for Anomaly Detection*. Mar. 17, 2020. URL: https://towardsdatascience.com/lstm-autoencoder-for-anomaly-detection-e1f4f2ee7ccf.

[22] Gábor Melis, Tomáš Kočiský, and Phil Blunsom. “Mogrifier LSTM”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=SJe5P6EYvS.

Appendix A

Code to find the threshold

A.1 Method 1

Implementation of the anomaly threshold algorithm from [14].

```python
def find_threshold(data, model_name, params):

    """
    Uses source N algorithm to find the optimal threshold,
    "Computing Anomaly Score Threshold with Autoencoders Pipeline"
    """

    train, test = Data.train_test_split(data, 0.5)

    st_assigned = False
    while True:
        # ae is our autoencoder.

        ae.fit(train, train, epochs=50, verbose=0, shuffle=True)
        errors = compute_errors(ae,test)

        if not st_assigned:
            st = np.min(errors)
            st_assigned = True

        si = 0.01*(np.max(errors) - np.min(errors))

        anomalies = test[errors>st+si]

        if anomalies.shape[0] == test.shape[0]:
            # all samples are anomalies -> terminate
            st = 0.5 * (st + np.min(errors))
            return st
        elif anomalies.shape[0] == 0 and test.shape[0] != 0:
            # terminate if samples contains values but no anomalies
            return -1
        else:
            train = np.concatenate((train, anomalies))
            test = test[errors <= st + si]
            st = st + si
```

A.2 Method 2

Given that we have the training errors in `errors_train`, we can use the approach suggested in [6] and treat the largest error in the train set as the decision boundary:

```python
np.max(errors_train)
```

A.3 Method 3

Given that we have the training errors in `errors_train`, we can find the largest error using a histogram in NumPy as follows:

```python
np.histogram(errors_train)[1][-2]
```

Appendix B

Architecture

The following chapter lists the network architectures used in the experiments.

B.1 LSTM based autoencoders

B.1.1 LSTM_AE

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 100, 8)            0         
_________________________________________________________________
lstm_1 (LSTM)                (None, 100, 32)           5248      
_________________________________________________________________
lstm_2 (LSTM)                (None, 16)                3136      
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 100, 16)           0         
_________________________________________________________________
lstm_3 (LSTM)                (None, 100, 16)           2112      
_________________________________________________________________
lstm_4 (LSTM)                (None, 100, 32)           6272      
_________________________________________________________________
time_distributed_1 (TimeDist (None, 100, 8)            264       
=================================================================
Total params: 17,032
Trainable params: 17,032
Non-trainable params: 0
_________________________________________________________________
```

B.1.2 LSTM_AE_Small

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         (None, 100, 8)            0         
_________________________________________________________________
lstm_5 (LSTM)                (None, 100, 32)           5248      
_________________________________________________________________
lstm_6 (LSTM)                (None, 3)                 432       
_________________________________________________________________
repeat_vector_2 (RepeatVecto (None, 100, 3)            0         
_________________________________________________________________
lstm_7 (LSTM)                (None, 100, 3)            84        
_________________________________________________________________
lstm_8 (LSTM)                (None, 100, 32)           4608      
_________________________________________________________________
```

```
time_distributed_2 (TimeDist (None, 100, 8)            264
=================================================================
Total params: 10,636
Trainable params: 10,636
Non-trainable params: 0
_________________________________________________________________
```

B.1.3 LSTM_AE_Deep

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 100, 8)            0
_________________________________________________________________
lstm_1 (LSTM)                (None, 100, 32)           5248
_________________________________________________________________
lstm_2 (LSTM)                (None, 100, 16)           3136
_________________________________________________________________
lstm_3 (LSTM)                (None, 8)                 800
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 100, 8)            0
_________________________________________________________________
lstm_4 (LSTM)                (None, 100, 8)            544
_________________________________________________________________
lstm_5 (LSTM)                (None, 100, 16)           1600
_________________________________________________________________
lstm_6 (LSTM)                (None, 100, 32)           6272
_________________________________________________________________
time_distributed_1 (TimeDist (None, 100, 8)            264
=================================================================
Total params: 17,864
Trainable params: 17,864
Non-trainable params: 0
_________________________________________________________________
```

B.1.4 LSTM_AE_Deep2

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 100, 8)            0
_________________________________________________________________
lstm_7 (LSTM)                (None, 100, 32)           5248
_________________________________________________________________
lstm_8 (LSTM)                (None, 100, 16)           3136
_________________________________________________________________
lstm_9 (LSTM)                (None, 100, 8)            800
_________________________________________________________________
lstm_10 (LSTM)               (None, 4)                 208
_________________________________________________________________
repeat_vector_2 (RepeatVecto (None, 100, 4)            0
_________________________________________________________________
lstm_11 (LSTM)               (None, 100, 4)            144
_________________________________________________________________
lstm_12 (LSTM)               (None, 100, 8)            416
_________________________________________________________________
lstm_13 (LSTM)               (None, 100, 16)           1600
_________________________________________________________________
lstm_14 (LSTM)               (None, 100, 32)           6272
_________________________________________________________________
time_distributed_2 (TimeDist (None, 100, 8)            264
=================================================================
Total params: 18,088
Trainable params: 18,088
Non-trainable params: 0
_________________________________________________________________
```

B.2 TCN based autoencoders

B.2.1 TCN_AE

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 100, 8)            0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 100, 100)          1700
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 50, 100)           0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 50, 50)            10050
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 25, 50)            0
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 25, 25)            2525
_________________________________________________________________
max_pooling1d_3 (MaxPooling1 (None, 12, 25)            0
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 12, 12)            612
_________________________________________________________________
dense_4 (Dense)              (None, 12, 6)             78
_________________________________________________________________
up_sampling1d_1 (UpSampling1 (None, 24, 6)             0
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 24, 12)            156
_________________________________________________________________
up_sampling1d_2 (UpSampling1 (None, 48, 12)            0
_________________________________________________________________
conv1d_6 (Conv1D)            (None, 48, 25)            625
_________________________________________________________________
up_sampling1d_3 (UpSampling1 (None, 96, 25)            0
_________________________________________________________________
conv1d_7 (Conv1D)            (None, 96, 50)            2550
_________________________________________________________________
zero_padding1d_1 (ZeroPaddin (None, 100, 50)           0
_________________________________________________________________
conv1d_8 (Conv1D)            (None, 100, 100)          10100
_________________________________________________________________
time_distributed_4 (TimeDist (None, 100, 8)            808
=================================================================
Total params: 29,204
Trainable params: 29,204
Non-trainable params: 0
_________________________________________________________________
```

B.2.2 TCN_AE2

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 100, 8)            0
_________________________________________________________________
conv1d_9 (Conv1D)            (None, 100, 100)          1700
_________________________________________________________________
max_pooling1d_4 (MaxPooling1 (None, 50, 100)           0
_________________________________________________________________
conv1d_10 (Conv1D)           (None, 50, 50)            10050
_________________________________________________________________
max_pooling1d_5 (MaxPooling1 (None, 25, 50)            0
_________________________________________________________________
conv1d_11 (Conv1D)           (None, 25, 25)            2525
_________________________________________________________________
max_pooling1d_6 (MaxPooling1 (None, 12, 25)            0
_________________________________________________________________
conv1d_12 (Conv1D)           (None, 12, 12)            612
_________________________________________________________________
reshape_1 (Reshape)          (None, 144)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 144)               0
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 435
```

```
_________________________________________________________________
dense_6 (Dense)              (None, 144)               576
_________________________________________________________________
reshape_2 (Reshape)          (None, 12, 12)            0
_________________________________________________________________
up_sampling1d_4 (UpSampling1 (None, 24, 12)            0
_________________________________________________________________
conv1d_13 (Conv1D)           (None, 24, 12)            300
_________________________________________________________________
up_sampling1d_5 (UpSampling1 (None, 48, 12)            0
_________________________________________________________________
conv1d_14 (Conv1D)           (None, 48, 25)            625
_________________________________________________________________
up_sampling1d_6 (UpSampling1 (None, 96, 25)            0
_________________________________________________________________
conv1d_15 (Conv1D)           (None, 96, 50)            2550
_________________________________________________________________
zero_padding1d_2 (ZeroPaddin (None, 100, 50)           0
_________________________________________________________________
conv1d_16 (Conv1D)           (None, 100, 100)          10100
_________________________________________________________________
time_distributed_4 (TimeDist (None, 100, 8)            808
=================================================================
Total params: 30,281
Trainable params: 30,281
Non-trainable params: 0
_________________________________________________________________
```

B.2.3 TCN_AE_Small

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 100, 8)            0
_________________________________________________________________
conv1d_17 (Conv1D)           (None, 100, 100)          1700
_________________________________________________________________
max_pooling1d_7 (MaxPooling1 (None, 50, 100)           0
_________________________________________________________________
conv1d_18 (Conv1D)           (None, 50, 50)            10050
_________________________________________________________________
max_pooling1d_8 (MaxPooling1 (None, 25, 50)            0
_________________________________________________________________
conv1d_19 (Conv1D)           (None, 25, 25)            2525
_________________________________________________________________
max_pooling1d_9 (MaxPooling1 (None, 12, 25)            0
_________________________________________________________________
conv1d_20 (Conv1D)           (None, 12, 12)            612
_________________________________________________________________
max_pooling1d_10 (MaxPooling (None, 6, 12)             0
_________________________________________________________________
conv1d_21 (Conv1D)           (None, 6, 6)              150
_________________________________________________________________
dense_8 (Dense)              (None, 6, 3)              21
_________________________________________________________________
up_sampling1d_7 (UpSampling1 (None, 12, 3)             0
_________________________________________________________________
conv1d_22 (Conv1D)           (None, 12, 6)             42
_________________________________________________________________
up_sampling1d_8 (UpSampling1 (None, 24, 6)             0
_________________________________________________________________
conv1d_23 (Conv1D)           (None, 24, 12)            156
_________________________________________________________________
up_sampling1d_9 (UpSampling1 (None, 48, 12)            0
_________________________________________________________________
conv1d_24 (Conv1D)           (None, 48, 25)            625
_________________________________________________________________
up_sampling1d_10 (UpSampling (None, 96, 25)            0
_________________________________________________________________
conv1d_25 (Conv1D)           (None, 96, 50)            2550
_________________________________________________________________
zero_padding1d_3 (ZeroPaddin (None, 100, 50)           0
_________________________________________________________________
conv1d_26 (Conv1D)           (None, 100, 100)          10100
_________________________________________________________________
time_distributed_5 (TimeDist (None, 100, 8)            808
=================================================================
Total params: 29,339
Trainable params: 29,339
Non-trainable params: 0
_________________________________________________________________
```

B.2.4 TCN_AE_Small2

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_6 (InputLayer)         (None, 100, 8)            0         
_________________________________________________________________
conv1d_27 (Conv1D)           (None, 100, 100)          1700      
_________________________________________________________________
max_pooling1d_11 (MaxPooling (None, 50, 100)           0         
_________________________________________________________________
conv1d_28 (Conv1D)           (None, 50, 50)            10050     
_________________________________________________________________
max_pooling1d_12 (MaxPooling (None, 25, 50)            0         
_________________________________________________________________
conv1d_29 (Conv1D)           (None, 25, 25)            2525      
_________________________________________________________________
max_pooling1d_13 (MaxPooling (None, 12, 25)            0         
_________________________________________________________________
conv1d_30 (Conv1D)           (None, 12, 12)            612       
_________________________________________________________________
max_pooling1d_14 (MaxPooling (None, 6, 12)             0         
_________________________________________________________________
conv1d_31 (Conv1D)           (None, 6, 6)              150       
_________________________________________________________________
dense_10 (Dense)             (None, 6, 3)              21        
_________________________________________________________________
reshape_3 (Reshape)          (None, 18)                0         
_________________________________________________________________
dense_11 (Dense)             (None, 3)                 57        
_________________________________________________________________
dense_12 (Dense)             (None, 18)                72        
_________________________________________________________________
reshape_4 (Reshape)          (None, 6, 3)              0         
_________________________________________________________________
up_sampling1d_11 (UpSampling (None, 12, 3)             0         
_________________________________________________________________
conv1d_32 (Conv1D)           (None, 12, 6)             42        
_________________________________________________________________
up_sampling1d_12 (UpSampling (None, 24, 6)             0         
_________________________________________________________________
conv1d_33 (Conv1D)           (None, 24, 12)            156       
_________________________________________________________________
up_sampling1d_13 (UpSampling (None, 48, 12)            0         
_________________________________________________________________
conv1d_34 (Conv1D)           (None, 48, 25)            625       
_________________________________________________________________
up_sampling1d_14 (UpSampling (None, 96, 25)            0         
_________________________________________________________________
conv1d_35 (Conv1D)           (None, 96, 50)            2550      
_________________________________________________________________
zero_padding1d_4 (ZeroPaddin (None, 100, 50)           0         
_________________________________________________________________
conv1d_36 (Conv1D)           (None, 100, 100)          10100     
_________________________________________________________________
time_distributed_6 (TimeDist (None, 100, 8)            808       
=================================================================
Total params: 29,468
Trainable params: 29,468
Non-trainable params: 0
_________________________________________________________________
```

TRITA -EECS-EX-2020:223