Long Short-Term Memory neural networks and Support Vector Data Description for anomaly detection

Auteur : Keydener, Jimmy
Promoteur(s) : Heuchenne, Cédric
Faculté : Faculté des Sciences
Diplôme : Master en sciences mathématiques, à finalité spécialisée en statistique
Année académique : 2019-2020
URI/URL : http://hdl.handle.net/2268.2/9212

Université de Liège

Master's thesis

Long Short-Term Memory neural networks and Support Vector Data Description for anomaly detection

Author:
Jimmy Keydener

Supervisor:
Pr. Cédric Heuchenne

A thesis submitted in fulfillment of the requirements for the Master's degree in mathematics

Department of mathematics

Faculty of Sciences

Academic year 2019-2020

Acknowledgements

I would like to thank my supervisor Pr. Cédric Heuchenne who give me the opportunity to work on this topic. I thank him for letting me do this thesis as my own work and for pointing me in the right direction when necessary. This year has been filled with learning from the work done on this thesis.

I would also like to acknowledge Luisa Mennicken and Sophie Klenkenberg who agreed to reread this thesis in detail and allowed me to greatly improve its quality.

Finally, I must express my gratitude to my family, my friends and my girlfriend who have always supported me through my years of study, and especially through this last demanding year.

Contents

Introduction

Anomaly detection refers to the problem of finding rare patterns in data which raise suspicions because they do not comply with an expected behavior. We can consider different kinds of applications like intrusion detection, image processing, system health monitoring and sensor networks. For example, an anomalous pattern coming from sensors on a machine could mean that the machine is ready to break.

As discussed in [5], most of the current studies on anomaly detection do not consider recent/past events to detect possible new incoming outliers. The use of Long Short-Term Memory (LSTM) networks is then proposed to deal with time dependent data related with anomaly detection problems. In machine learning, a LSTM network is a special type of recurrent neural networks that uses both short and long-term dependencies from the data set.

A brief review of the studies related to applying LSTM to time series and anomaly detection issues is presented in [5]. In this context, [24] suggests to use stacked LSTM networks. Though, they assumed a multivariate Gaussian distribution for the error vectors, which is not often realistic.

In this master thesis, we will thus consider a LSTM-based prediction model for sensor readings coming from a pulp and paper manufacturing machine. Anomalies will then result from too large prediction errors.

The goal of Support Vector Data Description (SVDD) is to describe a realistic domain for the data, excluding superfluous space. The resulting boundary can then be used to detect outliers. The method is inspired by Support Vector Classifiers: kernel functions allow for flexible boundaries as in Support Vector Machines.

SVDD and a discrimination rule based on the assumption on normality for the errors will be compared. Our goal is to show that for some real world applications the Gaussian distribution for the errors cannot hold and that the need of a non-parametric data description using kernels is real.

In the first chapter, we will present how the generalization from classic neural networks to the Long Short-Term Memory networks is done.

Afterward, the second chapter will present mathematically the Support Vector Data Description. Before that we will present the one-class Support Vector Machine which has inspired the SVDD.

In the third chapter, we will describe how we can use both LSTM and SVDD for anomaly detection. Moreover, we will present how we tune the parameters of a LSTM network in practice.

Finally, in the fourth chapter we will apply the techniques described in the three previous chapters on a real world application.

The contribution of this master thesis is to show that non-parametric data description techniques are necessary for some real world applications. Moreover, we propose a combination of LSTM and SVDD which has never been used in this context.

Chapter 1

Long Short-Term Memory

Neural networks have been developed to solve non-linear classification and regression problems. When the data is time dependent a generalization of classic neural networks is used, recurrent neural networks (RNN). However, the RNN are unable of learning both long and short-time dependencies from the data. To solve this problem Hochreiter and Schmidhuber in *Long Short-Term Memory* ([17]) have developed a new recurrent architecture, the Long Short-Term Memory (LSTM).

Firstly, we will introduce non-recurrent neural networks. From there, we will present recurrent neural networks and show why they are unable to learn both long and short-time dependencies. Then, we will put forward a solution to this problem by introducing the naive idea behind the LSTM and finally present LSTM neural networks. In the last part, we will demonstrate the LSTM that we will use in practice.

1.1 Non-recurrent neural networks

In this section, we will define non-recurrent neural networks. First, we will describe their general structure, known as layers. Afterward, we will define how they are used to make prediction. Finally, we will explain how these networks are optimized.

1.1.1 The different layers

A neural network is made of layers: an input layer, some hidden layers and an output layer. The input layer does not perform any computation and is used to link the input data to the network. The hidden layers are parts of a black-box model producing non-linear combinations of the inputs. The output layer produces a last calculation adapting the output of the network to the considered task. We associate to each layer an activation function which induces the non-linearity of the network during the training procedure. Each layer is composed of several units called neurons; neurons from two consecutive

layers may be linked by an edge, and we label each of these edges by a coefficient called weight. Moreover, we associate to each unit a coefficient called bias. This structure is defined as a Multilayer Perceptron (MLP). A fully connected Multilayer Perceptron is a neural network where all units from one layer, but the output layer, are linked to all units from the next layer by an edge.

How Multilayer Perceptrons are mathematically defined?

- Let L be the number of layers, where the layer 1 is the input layer, the layer L is the output one and layers 2 to $L-1$ are hidden layers.
- Let S_l be the number of units (neurons) of layer l, for $1 \leq l \leq L$.
- Let $a_i^{(l)}$ be the activation (i.e the output) of the unit i from layer l, for $1 \leq l \leq L$, $1 \leq i \leq S_l$.
- Let $f^{(l)}$ be the activation function of layer l, for $1 < l \leq L$.
- Let $w_{i,j}^{(l)}$ be the weight of the edge from neuron j in layer l to neuron i in layer $l+1$ for $1 \leq l < L$, $1 \leq j \leq S_l$ and $1 \leq i \leq S_{l+1}$.
- Let $w_{i,0}^{(l)}$ be the bias of the neuron i in the layer $l+1$, for $1 \leq l < L$ and $1 \leq i \leq S_{l+1}$.

Given this structure it is possible to predict the output vector through this network.

1.1.2 Predict output vector

Using the notations from the previous section we can easily mathematically define how Multilayer Perceptrons predict outputs vectors.

Let $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^p \times \mathbb{R}^m$ be any point from the data set; $\boldsymbol{x}$ is the vector of inputs and $\boldsymbol{y}$ the target. If activation functions and weights are set, the output vector $\hat{\boldsymbol{y}}$ predicted by the neural network can be calculated by the following formulas: we define the current i^{th} net input of a layer l, $i = 1, \ldots, S_l$, $l = 2, \ldots, L$ by

$$net_i^{(l)} = \sum_{j=0}^{S_{l-1}} w_{i,j}^{(l-1)} a_j^{(l-1)}.$$

From there, we can compute for any unit of any layer its activation; we have

$$\begin{aligned} a_0^{(l)} &= 1,\ l = 1, \ldots, L, \\ a_i^{(1)} &= x_i,\ i = 1, \ldots, p, \\ a_i^{(l)} &= f^{(l)}(net_i^{(l)}),\ l = 2, \ldots, L,\ i = 1, \ldots, S_l. \end{aligned} \tag{1.1}$$

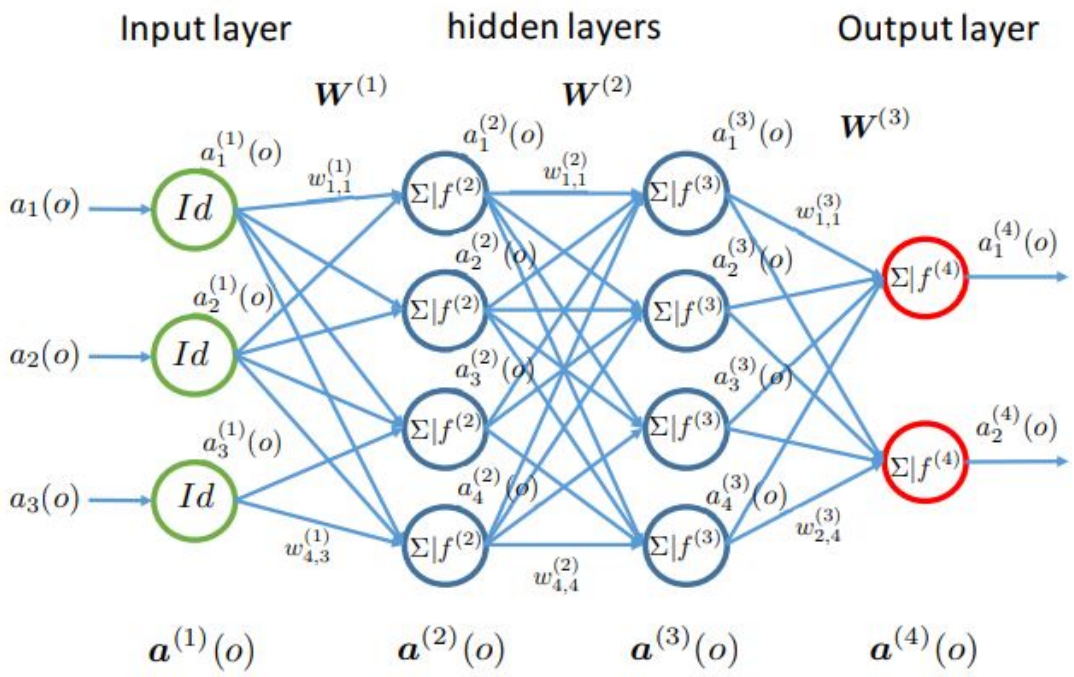


Figure 1.1 – *Representation of a Multilayer Perceptron, both hidden layers have four hidden units. The input layer has 3 units while the output one has 2 units. This image has been taken from [42].*

The predicted vector $\hat{\boldsymbol{y}}$ is then $\boldsymbol{a}^L$. The dimension of the input vector determines the number of neurons in the input layer. For a graphical representation of a Multilayer Perceptron, see Figure 1.1.

We still need to define how to choose the activation functions and the weights. The weights are found by an iterative optimization procedure, while activation functions are chosen in relation to the task in question.

1.1.3 Activation functions

Activation functions for the hidden layers are generally "S" shaped curves, such as the *logistic* (or *sigmoid*) function or *tanh*. In modern neural networks, we often consider the $ReLU$ activation function to avoid the vanishing gradient problem. We will discuss on this problem in the section 1.2.2. When we try to solve a m-dimensional regression problem, the output layer has m units and the activation functions are considered to be the identity. If the problem is a K-classification problem, the *softmax* function is commonly used for the K units of the output layer. Moreover, if all the activation functions for the hidden layers are defined to be the identity, the model collapses to a linear regression or multilogit model.

1.1.4 Optimization of the weights

In contrast to linear regression, it is not possible to find a deterministic optimal solution for neural networks. Therefore, the optimization of neural networks is done in an iterative way. Intuitively, we want to update the parameters in order to minimize a loss function.

For each weight in the network, we need to compute the gradient of the loss function with respect to this weight. Afterward, we can update this weight following the direction of the computed gradient. Although the calculation of an analytical formula for the gradient is trivial, evaluating it numerically is a thorn in the foot. Thus, we need to introduce the back-propagation algorithm which traverses the neural network in a backward way to calculate all the gradients.

Consider a loss function E, this function depends on the current weights $\boldsymbol{W}$, the target $\boldsymbol{y}$ and the predicted value $\hat{\boldsymbol{y}}$. We set $E(\boldsymbol{P}) = E(\boldsymbol{W}, \boldsymbol{x}, \hat{\boldsymbol{y}})$ to simplify notations. We assume that E is differentiable with respect to the targets.

For any weight $w_{i,j}^{(l)}$, we seek

$$\frac{\delta E(\boldsymbol{P})}{\delta w_{i,j}^{(l)}}. \tag{1.2}$$

However, we cannot calculate this value directly, thus we apply the chain rule and we get

$$\frac{\partial E(\boldsymbol{P})}{\partial w_{i,j}^{(l)}} = \frac{\partial E(\boldsymbol{P})}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial net_i^{(l+1)}} \frac{\partial net_i^{(l+1)}}{\partial w_{i,j}^{(l)}}. \tag{1.3}$$

For the two last factors we have

$$\begin{aligned} \frac{\partial a_i^{(l+1)}}{\partial net_i^{(l+1)}} &= f'(net_i^{(l+1)}), \\ \frac{\partial net_i^{(l+1)}}{\partial w_{i,j}^{(l)}} &= a_j^{(l)}. \end{aligned} \tag{1.4}$$

We can substitute these values into (1.3) and get

$$\frac{\partial E(\boldsymbol{P})}{\partial w_{i,j}^{(l)}} = \frac{\partial E(\boldsymbol{P})}{\partial a_i^{(l+1)}} f'(net_i^{(l+1)}) a_j^{(l)}. \tag{1.5}$$

By defining

$$\delta_i^{(l)} \equiv \frac{\partial E(\boldsymbol{P})}{\partial a_i^{(l)}} f'(net_i^{(l)}) = \frac{\partial E(\boldsymbol{P})}{\partial net_i^{(l)}}. \tag{1.6}$$

We reap

$$\frac{\partial E(\boldsymbol{P})}{\partial w_{i,j}^{(l)}} = \delta_i^{(l+1)} a_j^{(l)}. \tag{1.7}$$

Then, the calculation of the gradient comes back to calculate $\delta_i^{(l+1)}$. When developing the first factor of 1.6 we get

$$\begin{aligned}\frac{\partial E(\boldsymbol{P})}{\partial a_i^{(l)}} &= \sum_{j=0}^{S_{l+1}} \frac{\partial E(\boldsymbol{P})}{\partial net_j^{(l+1)}} \frac{\partial net_j^{(l+1)}}{\partial a_i^{(l)}} \\ &= \sum_{j=0}^{S_{l+1}} \frac{\partial E(\boldsymbol{P})}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial net_j^{(l+1)}} \frac{\partial net_j^{(l+1)}}{\partial a_i^{(l)}} \\ &= \sum_{j=0}^{S_{l+1}} \frac{\partial E(\boldsymbol{P})}{\partial a_j^{(l+1)}} f'(net_j^{(l+1)}) w_{j,i}^{(l)}.\end{aligned} \tag{1.8}$$

Thus, we find that $\delta_i^{(l)}$ is also defined by a recursive relation. Indeed,

$$\delta_i^{(l)} = \sum_{j=0}^{S_{l+1}} (\delta_j^{(l+1)} w_{j,i}^{(l)}) f'(net_i^{(l)}). \tag{1.9}$$

Moreover, we can compute

$$\delta_i^{(L)} = \frac{\partial E(\boldsymbol{P})}{\partial \boldsymbol{a}^{(L)}} f'(net_i^{(L)}), \tag{1.10}$$

since the loss function exclusively depends on the outputs of the network.

This implies that we can compute $\delta_i^{(l+1)}$ by using 1.9, and then the gradient. This procedure can be used to calculate the gradient with respect to any weight. As we should know the value of the activation for each layer, we need to perform a forward pass before being able to perform the backward pass.

Before being able to train the network, we need to set initial weights. Generally, they are randomly chosen close to 0. In this way, activation functions take small values as arguments. Since we always consider activation functions approximately linear in the neighborhood of 0, the initial model is close to a linear model, and the non-linearity is given by the training procedure. Now that we have shown how to compute the gradient for a single object, we need to define how to include it into the whole training procedure. We distinguish to independent choices: the update frequency (or batch size) and the optimizer.

Update frequency

The update frequency defines the number of individuals we will pass through the current network before updating the weights with respect to these individuals.

If we consider the stochastic weight update, the gradient is computed for the first individual of the learning sample with respect to the initial weights. Then, the weights are updated with respect to this gradient. Afterward, the gradient is computed for the second individual with respect to the **current** weights and so on. When the full data set has been treated, the first epoch is done. Since the training of neural networks is not deterministic,

it is generally valuable to train the network for more than one epoch. We define the initial weights of a new epoch by the final weights of the previous epoch.

We can also update the weights in batch: in contrast to the stochastic weight update the weights are only updated after each complete epoch. In other words, the network is not modified within an epoch. The gradient for any weight is computed as the average of the gradients computed for each individual.

In practice, we generally define an alternative frequency by using mini-batch: we define a fixed number of individuals say b. When the network has seen b objects we update the weights. We will discuss later how to choose the value for b.

Optimizer

The simplest optimizer is the gradient descent. In this algorithm we subtract the value of the current gradient from the current weights. Mathematically, let $\nabla_{w_{i,j}^{(l)}}$ be the gradient computed at any step for any update frequency, with respect to any weight $w_{i,j}^{(l)}$, with $1 \leq l < L$, $1 \leq j \leq S_l$ and $1 \leq i \leq S_{l+1}$. The new weight is computed by

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \gamma \cdot \nabla_{w_{i,j}^{(l)}}, \tag{1.11}$$

with $\gamma \in (0, 1)$, the learning rate. We present more developed optimizers in the appendix A.

1.2 From Multilayer Perceptron to Recurrent Neural Networks

1.2.1 Definition and notations

There exist a lot of generalizations of Multilayer Perceptron. For example, in order to deal with images, convolutional neural networks are very powerful. In this thesis, we will use recurrent neural networks (RNN). These networks are an adaptation of MLP built to deal with time dependent data. This adaptation allows the network to share parameters across different parts of the model. More precisely, a layer at a certain time may use outputs from the previous layer.

Let $\mathcal{N}$ be a RNN with n hidden units and a single recurrent layer, this layer is the counterpart of a hidden layer for Multilayer Perceptron. Moreover, let the data consists of T time steps. In this master thesis, we suppose time to be discrete (see [2] for a discussion on dynamical time data). Assume that an individual is defined by a p-dimensional vector of inputs for each of its T time steps. In contrast to a classic neural network, a RNN produces a vector of outputs for each time step. For any time step t, $t = 1, \ldots, T$, let

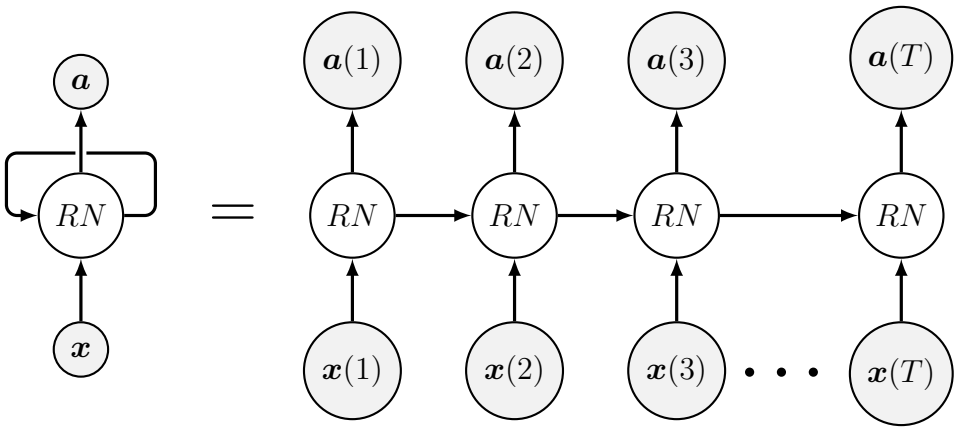


Figure 1.2 – *Representation of a folded and the same unfolded recurrent neural network; this network has only one recurrent layer with a self-recurrent connection.*

$\boldsymbol{x}^{ext}(t)$ be the p-dimensional vector of inputs and $\boldsymbol{a}(t)$ the m-dimensional vector of outputs at time t. Moreover, let $\boldsymbol{x}(t)$ be the concatenation of the vectors $\boldsymbol{a}(t-1)$ and $\boldsymbol{x}^{ext}(t)$, a $(m+p)$-dimensional vector. We set $\boldsymbol{a}(0) = \mathbf{0}$.

A common way to visualize a RNN is to unroll it with respect to the time steps. It allows us to consider the network as a simple MLP (see Figure 1.2). Let $\mathcal{N}^*$ be the unrolled network. $\mathcal{N}^*$ has one hidden layer for each time step t, $t = 1, \ldots, T$. Each of these layers receives as inputs $\boldsymbol{x}(t)$ and produces a n-dimensional vector of outputs $\boldsymbol{a}(t)$. The topology of $\mathcal{N}^*$ is defined as follows. For each unit in the recurrent layer of $\mathcal{N}$ a corresponding unit is created in the T hidden layers of $\mathcal{N}^*$. Moreover, the i^{th} unit of the layer t is connected to the j^{th} unit of the layer $t+1$ in $\mathcal{N}^*$ for all t in $\{1, \ldots, T-1\}$, if and only if, the unit i is connected to the unit j in $\mathcal{N}$, $i, j = 1, \ldots, m$.

We can see from Figure 1.2 that an unfolded RNN is indeed equivalent to a Multilayer Perceptron. However, the self-recurrent connection is the same for all time steps. As a consequence, the weight matrix must be equal for all hidden layers. More precisely, let $\boldsymbol{w}$ be the self-recurrent matrix of weights. This matrix contains a unique weight between each pair of units and also from each input line to each unit. We can incorporate this information into a $m \times (p+m+1)$ matrix, where the additional column holds for the bias.

In the same way as in the previous section, we can define the net input of unit i at time t:

$$net_i(t) = \sum_{j=0}^{m+p} w_{i,j} x_j(t), \tag{1.12}$$

$t = 1, \ldots, T$, $i = 1, \ldots, m$. From there, we define the vector of outputs at time t by:

$$a_i(t) = f_i(net_i(t)), \tag{1.13}$$

$t = 1, \ldots, T$. Where f_i denotes the activation function of the i^{th} unit, which is the same for all hidden layers, $i = 1, \ldots, m$.

Hence, the training of a RNN can be done by backpropagation under some constraints on the weights and some adjustments. A well-known and widely used algorithm to train RNN is back-propagation through time (BPTT). The philosophy of this algorithm is the same as simple backpropagation but adapted to time dependent network (for more details see [45]). We will derive the BPTT algorithm for a LSTM architecture in the section 1.3.6.

1.2.2 The vanishing and exploding gradient problems

In this section, we will present mathematical arguments to show how the vanishing gradient problem occurs, and why it stops RNN to learn long-time dependencies. Some proofs will be omitted and we refer to [16] or [17] for a deeper discussion on this problem.

Let suppose that our goal is to match some unit outputs at specific times with relative targets, often called *teachers signals*.

Let $S(t)$ be the set of indices i, $i \in \{1, \dots, m\}$, for which a specified target value $d_k(t)$ exists. The output of the i^{th} unit at time t should match as closely as possible this target, $t = 1, \dots, T$.

For any time step t, $t = 1, \dots, T$, we can define a m-dimensional vector $\boldsymbol{e}(t)$ by:

$$e_i(t) = \begin{cases} d_i(t) - a_i(t) & \text{if } i \in S(t), \\ 0 & \text{otherwise.} \end{cases}$$

The error at time t can be defined by:

$$E(t) = \frac{1}{2} \sum_{i=1}^{m} e_i(t)^2,$$

$t = 1, \dots, T$. We want to minimize the total error:

$$E = \sum_{\tau \in \{1,\dots,T\}} E(\tau).$$

We will now focus our study on a single time step t and consider the error at this time, i.e. $E(t)$. For any $k, k = 1, \dots, m$, this error produces the k^{th} unit back-propagated error signal defined by:

$$\delta_k(t) = \frac{\partial E(t)}{\partial net_k(t)}.$$

Intuitively, this error signal is the contribution of unit k to the total error at time t, and it is a generalization of δ defined for the MLP case.

Since BPTT is the same algorithm as the classic back-propagation adapted to RNN, the equations from the classic back-propagation are easily generalized to the BPTT case. Thus, equations 1.6 and 1.9 give

$$\delta_i(\tau) = f_i'(net_i(\tau))(\sum_{j=1}^{m} w_{j,i}\delta_j(\tau + 1)) = \frac{\partial E(t)}{\partial net_i(\tau)},\ 0 < \tau < t. \tag{1.14}$$

From equation 1.7, we can deduce that the contribution at time $\tau < t$ to the total weight update of $w_{j,i}$ is given by:

$$\gamma \delta_j(\tau) a_i(\tau - 1),$$

where $\gamma \in (0, 1)$ is the learning rate and $i \in \{1, \ldots, m\}$ is an arbitrary unit connected to unit j. We focus our study on a non-input weight, but a similar reasoning can be done on an input weight.

We want to prove that, for $\tau << t$, this contribution tends to 0. If this is true, only short time errors will contribute to the weight update. The network will therefore be unable to learn from long-term dependencies. Contrariwise, if this contribution tends to ∞, the network will learn only from long-term dependencies. Before we can prove this, we need some intermediate results.

For these results, we will focus on the error flow occurring at an arbitrary unit k at time t to a unit v at time $t - q$, $q > 0$, $q < t < T$, $1 < k, v < m$. This error is propagated back in time for q time steps. Intuitively, this is the contribution from the error signal at time t to the error signal at time $t - q$ and then is defined by:

$$\frac{\partial \delta_v(t - q)}{\partial \delta_k(t)}. \tag{1.15}$$

It has been shown in [17] that this local error flow can be computed with $l_q = v$ and $l_0 = k$ by

$$\frac{\partial \delta_v(t - q)}{\partial \delta_k(t)} = \sum_{l_1=1}^{m} \ldots \sum_{l_{q-1}=1}^{m} \prod_{i=1}^{q} f'_{l_i}(net_{l_i}(t - i)) w_{l_i, l_{i-1}}. \tag{1.16}$$

The proof can be done by induction and is omitted in this master thesis.

Intuitively if $|f'_{l_i}(net_{l_i}(t - i)) w_{l_i, l_{i-1}}| < 1 \, \forall i$, as q grows, the local error flow vanishes. Contrariwise, if $|f'_{l_i}(net_{l_i}(t - i)) w_{l_i, l_{i-1}}| > 1 \, \forall i$, as q grows, the local error flow explodes. For example, consider the *logistic* function: its derivative maximum value is 0.25. This implies that $|f'_{l_i}(net_{l_i}(t - i)) w_{l_i, l_{i-1}}| < 1$ if $|w_{l_i, l_{i-1}}| < 4 \, \forall i$. Furthermore, in [17] the authors have shown that increasing the weights is not a solution. For more details about this analysis we refer to [17] or [2].

Moreover, we can prove by induction on q using 1.12 along with 1.13 and 1.14 that

$$\frac{\partial \delta_v(t - q)}{\partial \delta_k(t)} = \frac{\partial net_k(t)}{\partial net_v(t - q)}. \tag{1.17}$$

The proof is done by using the chain rule and the reader is free do to it.

We can now turn our attention back to $\delta_i(\tau) a_j(\tau - 1)$, $\tau = 1, \ldots, t$. Keeping in mind that that our goal is to prove that this value tends to 0 when $\tau << t$. By definition and using 1.12 we have

$$\delta_i(\tau) a_j(\tau - 1) = \frac{\partial E(t)}{\partial net_i(\tau)} \frac{\partial net_i(\tau)}{\partial w_{i,j}}. \tag{1.18}$$

From there, by using the chain rule we get

$$\delta_i(\tau)a_j(\tau-1) = \sum_{k=1}^{m} \frac{\partial E(t)}{\partial net_k(t)} \frac{\partial net_k(t)}{\partial net_i(\tau)} \frac{\partial net_i(\tau)}{\partial w_{i,j}}. \tag{1.19}$$

From equations 1.16 and 1.17 considering the *logistic* function we deduce that for $\tau << t$, we have

$$\left| \frac{\partial net_k(t)}{\partial net_i(\tau)} \right| \to 0,\ k = 1, \ldots, m.$$

Then,

$$|\delta_i(\tau)a_j(\tau-1)| \to 0.$$

The weight $w_{i,j}$ is thus mainly updated with respect to short time error signals. This means that even if there might exist a change in $w_{i,j}$ leading the outputs to a better basin of attraction, the gradient of the error with respect to the weight does not clearly reflect that possibility. This is the vanishing gradient problem. We can explain the exploding gradient problem in the same way.

1.3 From Recurrent Neural Networks to Long Short-Term Memory

First, we will present the naive idea to avoid vanishing gradient problem. However, we will see that this idea is not applicable to a real-size network. Then, we will explain how to adapt the naive idea to a real recurrent neural network. This will lead us to the first Long Short-Term Memory architecture presented in [17]. Finally, we will explain how this architecture was adapted to solve another drawback, the internal state drift.

1.3.1 Naive idea

The naive idea behind the building of LSTM is to keep a constant error flow to avoid exploding and vanishing gradient problems. Consider a network which has only one unit j with a unique connection to itself. For any time $t < T$, the local error backflow of this unit according to the rules from the previous section (see 1.14) is

$$\delta_j(t) = f_j'(net_j(t))w_{j,j}\delta_j(t+1).$$

Thus, to ensure a constant error backflow, it is mandatory to have

$$f_j'(net_j(t))w_{j,j} = 1.$$

When we integrate the differential equation above, we get

$$f_j(net_j(t)) = \frac{net_j(t)}{w_{j,j}}. \tag{1.20}$$

Equation 1.20 implies that the function f_j has to be linear. Moreover, we need the activation of the unit to remain constant since

$$a_j(t+1) = f_j(net_j(t+1)) = f_j(w_{j,j}a_j(t)) = a_j(t).$$

If these two conditions are met, the constant error flow is achieved. We call this trick the central error carousel (CEC). In [17], they obtain the CEC by setting f_j equal to the identity function and the weight $w_{j,j}$ to 1.

1.3.2 Failures of the naive CEC

Obviously, real neural networks are not limited to a single unit, and the extension of the CEC to more units is not straightforward and invokes two problems: the input and the output weight conflicts.

The input weight conflict

For simplicity, consider that only a single input weight $w_{j,i}$ is added (this weight links an input unit i to the hidden unit j using the CEC). Assume that the total error can be reduced by simply training the network on the unit j in response to a certain input coming from i for a long period of time.

For example, given a sequence of characters, consider a system whose goal is to return 1 if the sequence is inside parentheses and 0 otherwise. Unit j needs to be active only when the relevant outputs (the characters inside the parentheses) are coming.

Then, $w_{j,i}$ has two conflicting roles. Its first role is to store the inputs, i.e. switching on j when needed. In this example, the unit j must be turned on when the input is "(". Its second role is to protect the inputs, i.e. protecting the inputs stored in j to be impacted by irrelevant inputs which will switch off j. Thus, the weight needs to prevent the sequence to turn off the unit if the character is different from ")". Keeping in mind that f_j is linear, this implies that the activation cannot be used to set to 0 an input which has entered into the network whereas it shouldn't.

This has as a consequence that the single weight $w_{j,i}$ will receive conflicting signals. Indeed, certain inputs will cause a reduction of the weight, because these inputs should not enter the network, while others will increase the weight. Thus, the value of $w_{j,i}$ will oscillate and it is nearly impossible for the network to learn in this situation. An additional feature to control the write operation in the network needs to be added.

The output weight conflict

Suppose that an additional output weight $w_{k,j}$ is added (which links the unit j to an output unit k). Assume that unit j stores some previous inputs; only some of these inputs are

relevant. Then at different times, $w_{k,j}$ will train either to access some relevant information in j or to avoid perturbations coming from irrelevant information in j. Similarly to the input weight conflict, $w_{k,j}$ will receive conflicting update signals. For instance, with many tasks the early training stages of the network may be enough to reduce short-time lag errors. However, in later training stages, even if the situation is under control, unit j may cause the training to add avoidable errors by attempting to reduce long-time lags. Again, an additional feature needs to be added, this time to control the read operation.

Of course, input and output weight conflicts do not only arise from long-time lags. However, their effects become very pronounced in this case. Indeed, for an increase in lag time, stored information must be protected for much longer. Moreover, as the training stages advance more and more correct outputs also require to be protected.

We need to adapt the naive CEC to control both the write and the read operations.

1.3.3 Solutions for naive CEC

In [17], the authors have proposed to add additional features to the naive CEC allowing a constant error flow through a special self-connected unit, without the problems of the naive approach.

They add a multiplicative input gate in order to control the write operation. Its role is to protect the memory contents stored in unit j from perturbation coming from irrelevant inputs. In the same way, an additional multiplicative output gate unit is defined to protect other units from perturbation by irrelevant memory storing in j. It controls the read operation.

These additional features define a more complex structure called a memory cell block. Each of these blocks can contain several units. A network is composed of many memory cell blocks divided into layers. Let c_j^v be the v^{th} unit of memory cell block j. We define an arbitrary topology for the network and we focus on a single memory block.

Each of this memory block is built around a linear unit with an identity self-connection, this is the CEC central feature. The unit c_j^v gets input from $net_{c_j^v}$, it also gets input from net_{in_j} (input coming from the input gate) and net_{out_j} (input coming from the output gate). The values net_{in_j} and net_{out_j} are shared for all units v in memory block j. Let f_{in_j} be the activation function of the j^{th} input gate and f_{out_j} the activation function of the j^{th} output gate. For any time $t \in \{1, \ldots, T\}$, we define the activation of the gates by:

$$a_{in_j} = f_{in_j}(net_{in_j}(t)); a_{out_j} = f_{out_j}(net_{out_j}(t)).$$

where

$$net_{in_j}(t) = \sum_u w_{in_j,u} a_u(t-1), \tag{1.21}$$

$$net_{out_j}(t) = \sum_u w_{out_j,u} a_u(t-1). \tag{1.22}$$

We also define

$$net_{c_j^v}(t) = \sum_u w_{c_j^v,u} a_u(t-1).$$

The summation index u stands for all possible input units as gate units, memory cells, or conventional hidden units if they are any. This choice of architecture and connection between the different units is up to the user. Moreover, all the units may use information about the current state of the network. For example, an input gate may use inputs from other memory cells to decide whether or not to store certain information. For any time $t \in \{1, \ldots, T\}$, we define the activation of c_j^v by

$$a_{c_j^v}(t) = a_{out_j}(t) h(s_{c_j^v}(t)),$$

where the internal state $s_{c_j^v}(t)$ is

$$s_{c_j^v}(0) = 0; s_{c_j^v}(t) = s_{c_j^v}(t-1) + a_{in_j}(t) g(net_{c_j^v}(t)).$$

The differentiable function g squashes $net_{c_j^v}$, while the differentiable function h scales cell outputs from the internal state $s_{c_j^v}$.

In summary, each cell has its own inputs, outputs and memory. Cells that belong to the same memory block, share input and output gates. This means that each cell might hold a different value in its memory, but the memory within one block is written to and read from all at once. Blocks sharing the layer that they receive inputs from and feed their outputs to, make up a layer.

The gates introduced before are used to avoid weight conflicts. The input gate controls the error flow from the input connections, while the output gate controls the error flow coming from the memory block. In other words, the input gate is used by the network to decide if it wants to keep or overwrite some information in the memory block. The output gate for its part decides when the network needs to access memory stocked in c_j^v and when to prevent other units from being perturbed by irrelevant information in c_j^v.

Error signal trapped into the CEC cannot change because of the identity function. However, different signals coming into the cell via the output gate may be superimposed. Then, the output gate needs to learn when to trap some error in its CEC by appropriately scaling them. Similarly, the input gate needs to know when to release error by scaling them. To resume, the multiplicative gates open and close access to constant error flow via the CEC. This is illustrated on Figure 1.3.

The learning algorithm used to train this type of network is more complex than what we have described in this master thesis. In [17], they proposed to use a fusion of a truncated BPTT (e.g., [43]) and a customized version of real time recurrent learning (RTRL). While the BPTT uses a forward pass in order to compute the outputs and then a backward pass to compute the derivative and update the weight, the RTRL uses only a single forward pass. Its goal is to calculate the derivative of states and outputs with respect to the weights as the network processes the sequence. This algorithm allows the network to update the

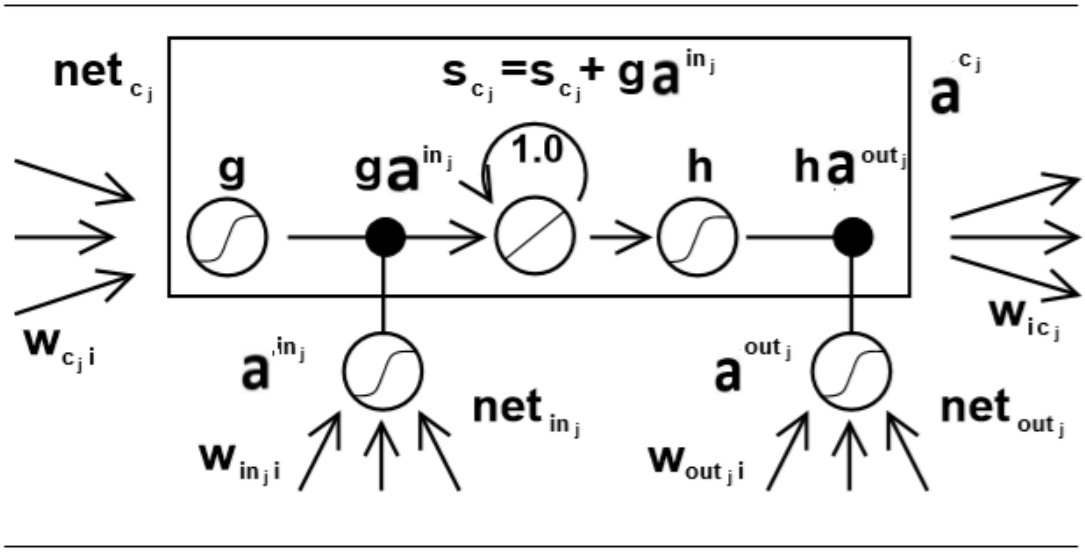


Figure 1.3 – *Representation of memory block as described by [17] in their paper. The self recurrent connection with weight equal to 1 is the central CEC feature with a linear identity function. The gate units control the access to the CEC. This image has been taken from [17].*

weights before full run. Its major drawback is the computational cost (for more details see [44]).

The goal of the LSTM learning algorithm is to not let the error get backpropagated except for the error on the memory state. Only the gradient of the cell is propagated back through time, and the gradient for the other recurrent connections is truncated. The BPTT part was truncated after one time step, because it was felt that long time dependencies would be dealt with by the memory blocks, and not by the (vanishing) flow of activation around the recurrent connections. The mathematical details are very technical and not valuable for this thesis. We refer the reader to [17] for a deeper mathematical analysis.

Moreover, these days we prefer to use a full gradient computation for LSTM. It will be described in section 1.3.6.

1.3.4 Internal state drift

This first ingenious LSTM architecture has a huge drawback, the internal state drift.

Suppose that h is defined as the *logistic* function (or any bounded activation function). If the inputs of the memory cell c_j^v are mostly positive (resp. negative), it will imply an increase (resp. decrease) of its internal state $s_{c_j^v}$ in an unbounded fashion. As a consequence, h will saturate. This saturation, on the one hand, will cause the gradient h' to vanish and thus block incoming errors (through the output gate). On the other hand, the

cell output will be equal to the output gate activation, and then, the memory cell will not work anymore as a memory.

1.3.5 Solutions to internal state drift

In [17], they partially solved this internal state drift problem by specifying precise initial values for some parameters. Later on, an alternative solution to this problem has been found in [12] by introducing a forget gate to the network architecture. This gate learns to reset memory blocks when their stocked information is out of date and not useful anymore. The reset is not brutal and can be done gradually.

To incorporate this gate into the memory block, we replace the unit weight from the CEC by the forget activation.

More precisely, let φ_j denote the j^{th} forget gate. We calculate its activation with

$$net_{\varphi_j}(t) = \sum_u w_{\varphi_j,u} a_u(t-1);\ a_{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)).$$

As a consequence, we update the equation for $s_{c_j^v}$ as

$$s_{c_j^v}(t) = a_{\varphi_j}(t) s_{c_j^v}(t-1) + a_{in_j}(t) g(net_{c_j^v}(t)).$$

In [12], they proposed to set f_{φ_j} as the *logistic* function. For a graphical representation of the newly created memory block, see Figure 1.4.

The addition of the forget gate to the network has as a consequence that the network will lose its constant error flow property. However, it has been shown empirically that this network is still able to learn long time dependencies efficiently. The learning algorithm for this newly created LSTM is barely the same than for the previous structure, we refer the reader to [12] for more details.

1.3.6 Vanilla LSTM in Keras

In this section, we will present the architecture that we will use in practice. In contrast to the two previous architectures presented, this architecture uses a full gradient computation to update the weights. Then, we cannot guarantee that the gradient will not vanish. However, when the input gate is close (its activation close to 0) and the forget gate is open (its activation close to 1), there is no new information written in the memory state and the network can keep learning long-term dependencies. Mathematically, we cannot prove that these gates will stay close (or open) when they need to. Though, in [13] they have shown that using the full gradient gave slightly higher performance than the original algorithm, and this gradient is more accurate than the truncated gradient. Thus, we choose to consider the full gradient computation for this master thesis.

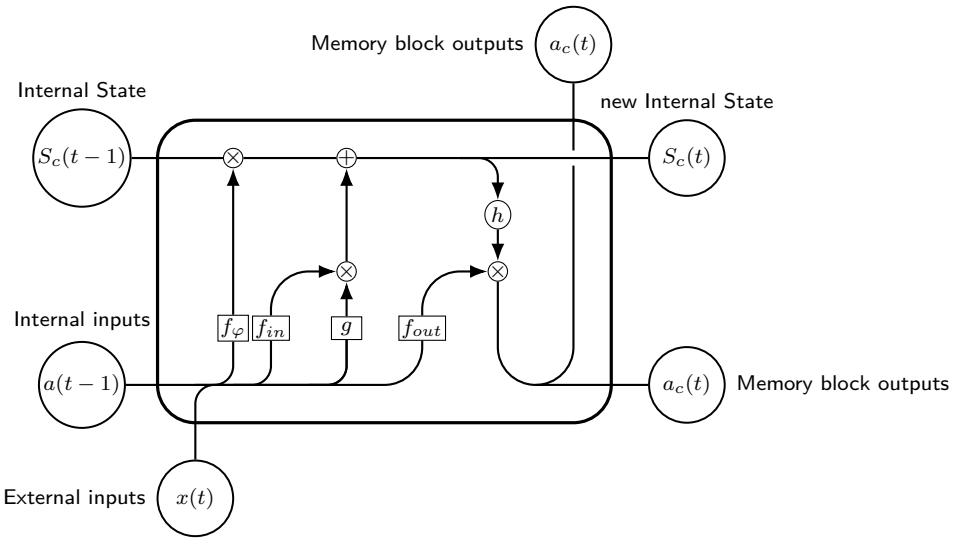


Figure 1.4 – *Representation of a memory block for a LSTM architecture. The indices distinguishing cells into the block are implied.* $\otimes$ *stands for an element-wise multiplication, while* $\oplus$ *stands for an element-wise addition. External inputs represent the inputs coming from the data set at time* t*. The internal inputs represent all inputs coming from the previous network time step; it may represent block and gate outputs. However, in most LSTM architectures, we set* $a(t-1) = a_c(t-1)$*.*

First, we will adapt our notations from the previous section to present the forward pass of the considered LSTM architecture. Afterward, we will derive the complete gradient of this architecture.

Assume a multi-dimensional regression problem, let $\{\boldsymbol{x}(1), \ldots, \boldsymbol{x}(T)\} \in \mathbb{R}^{p_1}$ be the input vectors, and $\boldsymbol{y} \in \mathbb{R}^{p_2}$ the target vector. Consider a network with one input layer, 2 hidden LSTM layers and one output layer. The extension to more LSTM hidden layers is straightforward. Let d_1 (resp. d_2) be the number of LSTM memory blocks of the first hidden layer (resp. second hidden layer). Consider that each memory block has a single memory cell and is connected to all memory blocks (including itself) in the same hidden layer.

For any $j \in \{1, \ldots, d_l\}$, $net_{c_j}^{(l)}(t)$, $net_{in_j}^{(l)}(t)$, $net_{out_j}^{(l)}(t)$, $net_{\varphi_j}^{(l)}(t)$ denotes respectively the net input of the cell, input, output, and forget gates at time t of memory block j from LSTM hidden layer l, $l \in \{1, 2\}$, $t \in \{1, \ldots, T\}$.

For any $m \in \{c, in, out, \varphi\}$, we set $\boldsymbol{net}_m^l(t) = (net_{m_1}^{(l)}(t), \ldots, net_{m_{d_l}}^{(l)}(t))^T$, $l \in \{1, 2\}$, $t \in \{1, \ldots, T\}$.

For any $j \in \{1, \ldots, d_l\}$, $a_{c_j}^{(l)}(t)$ is the output of memory block j from hidden layer l, $l \in \{1, 2\}$, $t \in \{0, \ldots, T\}$, where $a_{c_j}^{(l)}(0) = 0$. We set $\boldsymbol{a}_c^l(t) = (a_{c_1}^{(l)}(t), \ldots, a_{c_{d_l}}^{(l)}(t))^T$, $l \in \{1, 2\}$, $t \in \{0, \ldots, T\}$.

With these notations we can describe the forward pass of the network, we will introduce

other notations when needed.

Forward pass

For any $m \in \{c, in, out, \varphi\}$ and $t \in \{1, \dots, T\}$, the net inputs at time t of the first layer are computed with:

$$\boldsymbol{net}_m^{(1)}(t) = \boldsymbol{W}_m^{(1)}\boldsymbol{x}(t) + \boldsymbol{U}_m^{(1)}\boldsymbol{a}_c^{(1)}(t-1),$$

where $\boldsymbol{W}_m^{(1)} \in \mathbb{R}^{d_1 \times p_1}$ are the input weight matrices, and $\boldsymbol{U}_m^{(1)} \in \mathbb{R}^{d_1 \times d_1}$ are the recurrent weight matrices.

The activation is computed with:

$$a_{m_j}^{(1)}(t) = f_m^{(1)}(net_{m_j}^{(1)}(t)),$$

$j \in \{1, \dots, d_1\}$, $m \in \{in, out, \varphi\}$, $t \in \{1, \dots, T\}$. Where, $f_{in}^{(1)}$, $f_{out}^{(1)}$, $f_{\varphi}^{(1)}$ are respectively the activation functions of the input, output and forget gates.

We compute the internal state of each block as:

$$S_{c_j}^{(1)}(0) = 0;\ S_{c_j}^{(1)}(t) = a_{\varphi_j}^{(1)}(t)S_{c_j}^{(1)}(t-1) + a_{in_j}^{(1)}(t)g^{(1)}(net_{c_j}^{(1)}(t)),$$

$j \in \{1, \dots, d_1\}$, $t \in \{1, \dots, T\}$, where $g^{(1)}$ is a differentiable function.

The memory blocks outputs are defined by:

$$a_{c_j}^{(1)}(t) = a_{out_j}^{(1)}(t)h^{(1)}(S_{c_j}^{(1)}(t)),$$

$j \in \{1, \dots, d_1\}$, $t \in \{1, \dots, T\}$, where $h^{(1)}$ is a differentiable function.

When the forward pass for the first layer is done, we can compute it for the second hidden layer. This layer uses the output of the previous layer as input. More precisely, for any $m \in \{c, in, out, \varphi\}$ and $t \in \{1, \dots, T\}$, the net inputs at time t of the second layer are computed with:

$$\boldsymbol{net}_m^{(2)}(t) = \boldsymbol{W}_m^{(2)}\boldsymbol{a}_c^{(1)}(t) + \boldsymbol{U}_m^{(2)}\boldsymbol{a}_c^{(2)}(t-1),$$

where $\boldsymbol{W}_m^{(2)} \in \mathbb{R}^{d_2 \times d_1}$ are the input weight matrices, and $\boldsymbol{U}_m^{(2)} \in \mathbb{R}^{d_2 \times d_2}$ are the recurrent weight matrices.

The activation is computed with:

$$a_{m_j}^{(2)}(t) = f_m^{(2)}(net_{m_j}^{(2)}(t)),$$

$j \in \{1, \dots, d_2\}$, $m \in \{in, out, \varphi\}$, $t \in \{1, \dots, T\}$, where $f_{in}^{(2)}$, $f_{out}^{(2)}$, $f_{\varphi}^{(2)}$ are respectively the activation functions of the input, output and forget gates.

We compute the internal state of each block as:

$$S_{c_j}^{(2)}(0) = 0;\ S_{c_j}^{(2)}(t) = a_{\varphi_j}^{(2)}(t) S_{c_j}^{(2)}(t-1) + a_{in_j}^{(2)}(t) g^{(2)}(net_{c_j}^{(2)}(t)),$$

$j \in \{1, \ldots, d_2\}$, $t \in \{1, \ldots, T\}$, where $g^{(2)}$ is a differentiable function.

The memory blocks outputs are defined with:

$$a_{c_j}^{(2)}(t) = a_{out_j}^{(2)}(t) h^{(2)}(S_{c_j}^{(2)}(t)),$$

$j \in \{1, \ldots, d_2\}$, $t \in \{1, \ldots, T\}$, where $h^{(2)}$ is a differentiable function.

Finally, we can compute the predicted vector $\hat{\boldsymbol{y}}$ through the output layer with:

$$\hat{\boldsymbol{y}} = \boldsymbol{W}^{(3)} \boldsymbol{a}_c^{(2)}(T),$$

where $\boldsymbol{W}^{(3)} \in \mathbb{R}^{p_2 \times d_2}$ is the matrix of output weights.

Backward pass

For the backward pass we need to compute the derivatives from the output layer to the first hidden layer. We consider a squared error loss function defined by:

$$E = \frac{1}{2} \sum_{k=1}^{p_2} (\hat{y}_k - y_k)^2.$$

The generalization to other loss functions is straightforward. First, we need to update the weights of the output layer. We have

$$\begin{aligned} \frac{\partial E}{\partial (\boldsymbol{W}^{(3)})_{i,j}} &= \sum_{k=1}^{p_2} \frac{\partial E}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial (\boldsymbol{W}^{(3)})_{i,j}} \\ &= (\hat{y}_i - y_i) a_{c_j}^{(2)}(T), \end{aligned}$$

$i \in \{1, \ldots, p_2\}$, $j \in \{1, \ldots, d_2\}$. Which is sufficient to update the weights.

To calculate the gradient of the error with respect to the weights in the second LSTM hidden layer, we need to compute intermediate derivatives for each of the T time steps.

We start the computation at time $t = T$, and calculate recursively the derivatives while decrementing t to one. We first show how to compute the derivatives at time T, afterward we will show how to proceed for $t < T$. It will be sufficient to compute all the derivatives that are needed.

From the output layer we can compute

$$\begin{aligned} \delta_{a_{c_j}^{(2)}(T)} :&= \frac{\partial E}{\partial a_{c_j}^{(2)}(T)} \\ &= \sum_{k=1}^{p_2} \frac{\partial E}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_{c_j}^{(2)}(T)} \\ &= \sum_{k=1}^{p_2} (\hat{y}_k - y_k) (\boldsymbol{W}^{(3)})_{kj},\ j \in \{1, \ldots, d_2\}. \end{aligned}$$

It allows us to calculate

$$\begin{aligned}\delta_{S^{(2)}_{c_j}(T)} :&= \frac{\partial E}{\partial S^{(2)}_{c_j}(T)} \\ &= \frac{\partial E}{\partial a^2_{c_j}(T)} \frac{\partial a^{(2)}_{c_j}(T)}{\partial S^{(2)}_{c_j}(T)} \\ &= \delta_{a^{(2)}_{c_j}(T)} a^{(2)}_{out_j}(T)(h^{(2)})'(S^{(2)}_{c_j}(T)),\ j \in \{1, \dots, d_2\},\end{aligned}$$

and

$$\begin{aligned}\delta_{net^{(2)}_{out_j}(T)} :&= \frac{\partial E}{\partial net^{(2)}_{out_j}(T)} \\ &= \frac{\partial E}{\partial a^2_{c_j}(T)} \frac{\partial a^{(2)}_{c_j}(T)}{\partial a^{(2)}_{out_j}(T)} \frac{\partial a^{(2)}_{out_j}(T)}{\partial net^{(2)}_{out_j}(T)} \\ &= \delta_{a^{(2)}_{c_j}(T)} h^{(2)}(S^{(2)}_{c_j}(T))(f^{(2)}_{out})'(net^{(2)}_{out_j}(T)),\ j \in \{1, \dots, d_2\}.\end{aligned}$$

From there, we have

$$\begin{aligned}\delta_{net^{(2)}_{c_j}(T)} :&= \frac{\partial E}{\partial net^{(2)}_{c_j}(T)} \\ &= \frac{\partial E}{\partial S^{(2)}_{c_j}(T)} \frac{\partial S^{(2)}_{c_j}(T)}{\partial net^{(2)}_{c_j}(T)} \\ &= \delta_{S^{(2)}_{c_j}(T)} a^{(2)}_{in_j}(T)(g^{(2)})'(net^{(2)}_{c_j}(T)),\ j \in \{1, \dots, d_2\}.\end{aligned}$$

$$\begin{aligned}\delta_{net^{(2)}_{in_j}(T)} :&= \frac{\partial E}{\partial net^{(2)}_{in_j}(T)} \\ &= \frac{\partial E}{\partial S^{(2)}_{c_j}(T)} \frac{\partial S^{(2)}_{c_j}(T)}{\partial a^{(2)}_{in_j}(T)} \frac{\partial a^{(2)}_{in_j}(T)}{\partial net^{(2)}_{in_j}(T)} \\ &= \delta_{S^{(2)}_{c_j}(T)} g^{(2)}(net^{(2)}_{c_j}(T))(f^{(2)}_{in})'(net^{(2)}_{in_j}(T)),\ j \in \{1, \dots, d_2\}.\end{aligned}$$

$$\begin{aligned}\delta_{net^{(2)}_{\varphi_j}(T)} :&= \frac{\partial E}{\partial net^{(2)}_{\varphi_j}(T)} \\ &= \frac{\partial E}{\partial S^{(2)}_{c_j}(T)} \frac{\partial S^{(2)}_{c_j}(T)}{\partial a^{(2)}_{\varphi_j}(T)} \frac{\partial a^{(2)}_{\varphi_j}(T)}{\partial net^{(2)}_{\varphi_j}(T)} \\ &= \delta_{S^{(2)}_{c_j}(T)} S^{(2)}_{c_j}(T-1)(f^{(2)}_{\varphi})'(net^{(2)}_{\varphi_j}(T)),\ j \in \{1, \dots, d_2\}.\end{aligned}$$

Consider $t < T$, we have

$$\begin{aligned}
\delta_{a_{c_j}^{(2)}(t)} &:= \frac{\partial E}{\partial a_{c_j}^{(2)}(t)} \\
&= \sum_{m\in\{c,in,out,\varphi\}} \sum_{i=1}^{d_2} \frac{\partial E}{\partial net_{m_i}^{(2)}(t+1)} \frac{\partial net_{m_i}^{(2)}(t+1)}{\partial a_{c_j}^{(2)}(t)} \\
&= \sum_{m\in\{c,in,out,\varphi\}} \sum_{i=1}^{d_2} \delta_{net_{m_i}^{(2)}(t+1)} (\boldsymbol{U}_m^{(2)})_{i,j},\ j \in \{1,\ldots,d_1\}.
\end{aligned}$$

Moreover, since $S_{c_j}^{(2)}(t)$ receives also a backward gradient from $S_{c_j}^{(2)}(t+1)$, we reap

$$\begin{aligned}
\delta_{S_{c_j}^{(2)}(t)} &:= \frac{\partial E}{\partial S_{c_j}^{(2)}(t)} \\
&= \frac{\partial E}{\partial a_{c_j}^{(2)}(t)} \frac{\partial a_{c_j}^{(2)}(t)}{\partial S_{c_j}^{(2)}(t)} + \frac{\partial E}{\partial S_{c_j}^{(2)}(t+1)} \frac{\partial S_{c_j}^{(2)}(t+1)}{\partial a_{c_j}^{(2)}(t)} \\
&= \delta_{a_{c_j}^{(2)}(t)} a_{out_j}^{(2)}(t) (h^{(2)})'(S_{c_j}^{(2)}(t)) + \delta_{S_{c_j}^{(2)}(t+1)} \frac{\partial S_{c_j}^{(2)}(t+1)}{\partial S_{c_j}^{(2)}(t)}.
\end{aligned} \tag{1.23}$$

We have

$$\begin{aligned}
\frac{\partial S_{c_j}^{(2)}(t+1)}{\partial S_{c_j}^{(2)}(t)} &= a_{\varphi_j}^{(2)}(t+1) + \frac{\partial a_{\varphi_j}^{(2)}(t+1)}{\partial S_{c_j}^{(2)}(t)} S_{c_j}^{(2)}(t) \\
&+ a_{in_j}^{(2)}(t+1) \frac{\partial g^{(2)}(net_{c_j}^{(2)}(t+1))}{\partial S_{c_j}^{(2)}(t)} + \frac{\partial a_{in_j}^{(2)}(t+1)}{\partial S_{c_j}^{(2)}(t)} g^{(2)}(net_{c_j}^{(2)}(t+1)) \\
&= a_{\varphi_j}^{(2)}(t+1) + S_{c_j}^{(2)}(t) (f_{\varphi_j}^{(2)})'(net_{\varphi_j}^{(2)}(t+1)) (\boldsymbol{U}_{\varphi}^{(2)})_{jj} a_{out_j}^{(2)}(t) (h^{(2)})'(S_{c_j}^{(2)}(t)) \\
&+ a_{in_j}^{(2)}(t+1) (g^{(2)})'(net_{c_j}^{(2)}(t+1)) (\boldsymbol{U}_{c}^{(2)})_{jj} a_{out_j}^{(2)}(t) (h^{(2)})'(S_{c_j}^{(2)}(t)) \\
&+ g^{(2)}(net_{c_j}^{(2)}(t+1)) (f_{in_j}^{(2)})'(net_{in_j}^{(2)}(t+1)) (\boldsymbol{U}_{in}^{(2)})_{jj} a_{out_j}^{(2)}(t) (h^{(2)})'(S_{c_j}^{(2)}(t)).
\end{aligned} \tag{1.24}$$

Setting 1.24 into 1.23, we get

$$\begin{aligned}
\delta_{S_{c_j}^{(2)}(t)} &= \delta_{S_{c_j}^{(2)}(t+1)} a_{\varphi_j}^{(2)}(t+1) \\
&+ a_{out_j}^{(2)}(t) (h^{(2)})'(S_{c_j}^{(2)}(t)) \left(\delta_{a_{c_j}^{(2)}(t)} + \sum_{m\in\{c,in,\varphi\}} (\boldsymbol{U}_m^{(2)})_{jj} \delta_{net_{m_j}^{(2)}(t+1)} \right).
\end{aligned}$$

Finally, the values for $\delta_{net_{m_j}^{(2)}(t)}$ are computed by replacing T by t in the formulas for time step T, $j \in \{1,\ldots,d_2\}$, $m \in \{c, in, out, \varphi\}$.

When we have computed the derivatives for each time step, we can compute the gradient of a weight from the input weight matrices by

$$\begin{aligned}\frac{\partial E}{\partial (\boldsymbol{W}_m^{(2)})_{ij}} &= \sum_{t=1}^{T}\sum_{l=1}^{d_2} \frac{\partial E}{\partial net_{m_l}^{(2)}(t)} \frac{\partial net_{m_l}^{(2)}(t)}{\partial (\boldsymbol{W}_m^{(2)})_{ij}} \\ &= \sum_{t=1}^{T} \delta_{net_{m_i}^{(2)}(t)} a_{c_j}^{(1)}(t),\end{aligned}$$

$m \in \{c, in, out, \varphi\}$, $i \in \{1, \dots, d_2\}$, $j \in \{1, \dots, d_1\}$. Similarly, for the recurrent weight matrices we have

$$\begin{aligned}\frac{\partial E}{\partial (\boldsymbol{U}_m^{(2)})_{ij}} &= \sum_{t=1}^{T}\sum_{l=1}^{d_2} \frac{\partial E}{\partial net_{m_l}^{(2)}(t)} \frac{\partial net_{m_l}^{(2)}(t)}{\partial (\boldsymbol{U}_m^{(2)})_{ij}} \\ &= \sum_{t=1}^{T} \delta_{net_{m_i}^{(2)}(t)} a_{c_j}^{(2)}(t-1),\end{aligned}$$

$m \in \{c, in, out, \varphi\}$, $i \in \{1, \dots, d_2\}$, $j \in \{1, \dots, d_2\}$

We will now show how to compute these derivatives for the first layer. We simply need to adjust the gradient of the cell outputs with respect to the dependency between the 2 layers. More precisely, we have

$$\begin{aligned}\delta_{a_{c_j}^{(1)}(t)} &:= \frac{\partial E}{\partial a_{c_j}^{(1)}(t)} \\ &= (1 - \delta^{tT}) \sum_{m \in \{c,in,out,\varphi\}} \sum_{i=1}^{d_1} \frac{\partial E}{\partial net_{m_i}^{(1)}(t+1)} \frac{\partial net_{m_i}^{(1)}(t+1)}{\partial a_{c_j}^{(1)}(t)} \\ &+ \sum_{m \in \{c,in,out,\varphi\}} \sum_{i=1}^{d_2} \frac{\partial E}{\partial net_{m_i}^{(2)}(t)} \frac{\partial net_{m_i}^{(2)}(t)}{\partial a_{c_j}^{(1)}(t)} \\ &= (1 - \delta^{tT}) \sum_{m \in \{c,in,out,\varphi\}} \sum_{i=1}^{d_1} \delta_{net_{m_i}^{(1)}(t+1)} (\boldsymbol{U}_m^{(1)})_{i,j} \\ &+ \sum_{m \in \{c,in,out,\varphi\}} \sum_{i=1}^{d_2} \delta_{net_{m_i}^{(2)}(t)} (\boldsymbol{W}_m^{(2)})_{i,j},\end{aligned}$$

$j \in \{1, \dots, d_1\}$, where δ^{ab} is the kronecker delta (1 if a=b, 0 otherwise). The values for $\delta_{net_{m_j}^{(1)}(t)}$ and $\delta_{S_{c_j}^{(1)}(t)}$ are computed with the same formulas as for the second layer (by adjusting the necessary indices and values), $j \in \{1, \dots, d_2\}$, $m \in \{c, in, out, \varphi\}$, $t \in \{1, \dots, T\}$.

Finally, we can compute the gradient of the weight matrices in the same way than for

the second hidden layer. We have

$$\begin{aligned}\frac{\partial E}{\partial (\boldsymbol{W}_m^{(1)})_{ij}} &= \sum_{t=1}^{T}\sum_{l=1}^{d_1} \frac{\partial E}{\partial net_{m_l}^{(1)}(t)} \frac{\partial net_{m_l}^{(1)}(t)}{\partial (\boldsymbol{W}_m^{(1)})_{ij}} \\ &= \sum_{t=1}^{T} \delta_{net_{m_i}^{(1)}(t)} x_j(t),\end{aligned}$$

$m \in \{c, in, out, \varphi\}$, $i \in \{1, \dots, d_1\}$, $j \in \{1, \dots, p_1\}$.

And,

$$\begin{aligned}\frac{\partial E}{\partial (\boldsymbol{U}_m^{(1)})_{ij}} &= \sum_{t=1}^{T}\sum_{l=1}^{d_1} \frac{\partial E}{\partial net_{m_l}^{(1)}(t)} \frac{\partial net_{m_l}^{(1)}(t)}{\partial (\boldsymbol{U}_m^{(1)})_{ij}} \\ &= \sum_{t=1}^{T} \delta_{net_{m_i}^{(1)}(t)} a_{c_j}^{(1)}(t-1),\end{aligned}$$

$m \in \{c, in, out, \varphi\}$, $i \in \{1, \dots, d_1\}$, $j \in \{1, \dots, d_1\}$, which concludes the backward pass.

1.3.7 Variants, developments and drawbacks

From that point, the development of the LSTM was incremental. In [30], the author proposes to add new connections between the units in the memory block, called *peephole*. In the past years, many variations of LSTM have been introduced (for example in [9] and [1]). However, in [14] the authors have found that all these alternative structures were not performing significantly better than the vanilla LSTM.

In [10], a simplified version of LSTM called the *gated recurrent unit* (GRU) is presented. This method drops the output gate and reduces the number of parameters. More recently, in [22] a new recurrent network dealing with long-time dependencies is introduced, the *independently Recurrent neural network* (IndRNN). It allows to stack more layers without impacting negatively the performance of the network, in contrast to LSTM. They have proved empirically that this type of network can deal with much longer sequences than both GRU and LSTM. Moreover, in [26] the author has proved that hessian free optimization can deal with long-term dependencies.

Despite of all the recent researches on LSTM-like architecture, the vanilla LSTM is still a popular solution in the literature.

Even if the LSTM has a lot of advantages, it has also some drawbacks. The saturation of the function h has been solved, but this solution has a cost in terms of power for long-term dependencies learning. Moreover, the computational cost to train a LSTM is heavier than for a classic RNN.

Chapter 2

Support Vector Data Description

In this chapter, we will present a data domain description technique to deal with anomaly detection, the Support Vector Data Description (SVDD). The goal of the SVDD is to construct a minimal volume hypersphere containing all the non-outlier data points, excluding the superfluous space. This technique is robust with respect to outliers. By introducing kernel functions, we can map the data into another feature space and make the data description more flexible. In the literature, it is common to combine both LSTM and a discrimination rule to detect outliers. However, normality of the errors is often assumed. By using SVDD we do not make any assumption on this distribution. SVDD is inspired by the one-class Support Vector Machine (SVM) presented in [38]. We can prove under some constraints that one-class SVM and SVDD are equivalent.

First, we will set some reminders on the convex optimization theory. Afterward, we will briefly describe the one-class SVM. Finally, we will present in details the support vector data description.

2.1 Reminders on convex optimization, strong duality, and KKT conditions

This section will set some reminders on the convex optimization theory, the strong duality theorem and the Karush-Kuhn-Tucker conditions. We will however not give any proof, we refer to [6] for more details.

An optimization problem is called to be convex if it is of the form,

$$\begin{aligned} &\min f(\boldsymbol{w}), \\ &\text{subject to } g_i(\boldsymbol{w}) \leq 0,\ i = 1, \dots, m, \\ &\text{and } h_i(\boldsymbol{w}) = 0,\ i = 1, \dots, l, \end{aligned} \tag{2.1}$$

where $f, g_1, \ldots, g_m$ are convex functions and $h_1, \ldots, h_l$ are affine functions. If the solution of the problem is in a high dimensional space it might be difficult to solve this problem without using the Lagrange dual problem. This problem is of the form

$$\begin{aligned} max\ & g(\boldsymbol{\alpha}, \boldsymbol{\mu}) \\ \text{subject to } & \alpha_i \geq 0,\ i = 1, \ldots, m, \\ & \mu_i \geq 0,\ i = 1, \ldots, l, \end{aligned} \tag{2.2}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^m$ and $\boldsymbol{\mu} \in \mathbb{R}^l$ are the Lagrange multipliers and $g(\boldsymbol{\alpha}, \boldsymbol{\mu})$ is the Lagrange dual function. This function is defined by

$$g(\boldsymbol{\alpha}, \boldsymbol{\mu}) = \inf_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\mu}) = \inf_{\boldsymbol{w}} \left(f(\boldsymbol{w}) + \sum_{i=1}^{m} \alpha_i g_i(\boldsymbol{w}) + \sum_{i=1}^{l} h_i(\boldsymbol{w}) \right).$$

We call the problem 2.2 the dual problem, whereas 2.1 is the primal problem. If we denote by d^* the optimal value of the dual and by $p*$ the optimal value of the primal. We always have the weak duality property:

$$d^* \leq p^* \tag{2.3}$$

The duality gap is given by $p^* - d^*$. However, if we want to solve the primal problem through the dual, we need the strict equality to hold. We call this property the strong duality. When the optimization problem is convex, we can prove that any local optimal point is also a globally optimal point.

Theorem 2.1 (Boyd and Vandenberghe, 2004, Section 5.5.3)**.** *Consider any optimization problem where both the objective and the constraints are differentiable. Let $\boldsymbol{w}^*$ and $(\boldsymbol{\alpha}^*, \boldsymbol{\mu}^*)$ be any primal and dual optimal solutions respectively associated with a 0 duality gap. Then, we have the Karush-Kuhn-Tucker (KKT) conditions:*

$$\begin{aligned} g_i(\boldsymbol{w}^*) &\leq 0,\ i = 1, \ldots, m, \\ h_i(\boldsymbol{w}^*) &= 0,\ i = 1, \ldots, l \\ \alpha_i^* &\geq 0,\ i = 1, \ldots, m, \\ \alpha_i^* g_i(\boldsymbol{w}^*) &= 0,\ i = 1, \ldots, m, \\ \frac{\partial \mathcal{L}(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial \boldsymbol{w}}|_{(\boldsymbol{w}^*, \boldsymbol{\alpha}^*, \boldsymbol{\mu}^*)} &= 0 \end{aligned} \tag{2.4}$$

They are necessary conditions for optimality if the strong duality holds. Moreover, if the optimization problem is convex, they are sufficient conditions for optimality.

Theorem 2.2 (Boyd and Vandenberghe, 2004, Section 5.2.3, Refined Slater's condition)**.** *For any set of functions $f, g_1, \ldots, g_m$, and any set S, we define:*

$$\boldsymbol{D} \equiv (\cap_{i=1}^{m} domain(g_i)) \cap (domain(f))$$

and

$$relint(S) \equiv \{\boldsymbol{w} \in S \mid \exists r > 0, B(r, \boldsymbol{w}) \cap \mathit{aff}(S) \subset S\},$$

where $B(r, \boldsymbol{w})$ *is the open ball of center* $\boldsymbol{w}$ *and radius* r*, and* $\mathit{aff}(S)$ *is the affine hull of* S*. If we consider a convex optimization problem of the form 2.1, and if the first* k *constraint functions* $g_1, \ldots, g_k$ *are affine, then strong duality for problem 2.1 holds if there exists a* $\boldsymbol{w} \in relint(\boldsymbol{D})$ *such that:*

$$\begin{aligned} g_i(\boldsymbol{w}) &\leq 0,\ i = 1, \ldots, k, \\ g_i(\boldsymbol{w}) &< 0,\ i = k+1, \ldots, m, \\ h_i(\boldsymbol{w}) &= 0,\ i = 1, \ldots, l. \end{aligned}$$

2.2 One-class Support Vector Machine

In this section, we will briefly explain what is the one-class SVM (for more details we refer the reader to [38]). The goal of the one-class SVM is to find a maximum margin hyperplane in a feature space which separates the data set from the origin. This algorithm returns a function f that takes the value "1" for most of the points but in a small region and takes the value "-1" elsewhere. We can map the data point into a feature space in order to have a better description of the points. Let $\Phi : \mathbb{R}^n \to F$ be such a feature map, where F is a dot product space. We choose F such that the dot product in the space defined by Φ can be computed by a simple kernel function

$$\Phi(\boldsymbol{x})^T \Phi(\boldsymbol{y}) = k(\boldsymbol{x}, \boldsymbol{x}).$$

For example, we can consider the Gaussian kernel defined by:

$$k(\boldsymbol{x}, \boldsymbol{x}) = \exp\left(\frac{||\boldsymbol{x} - \boldsymbol{x}||^2}{2\sigma^2}\right),\ \sigma \in \mathbb{R}.$$

We consider the data set

$$\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^p.$$

By using some geometric arguments, it is possible to show (the details are omitted in this thesis) that to find a hyperplane discriminating the data by a maximum margin we can consider the following problem:

$$\min_{\boldsymbol{w} \in F,\, \boldsymbol{\xi} \in \mathbb{R}^n,\, \rho \in \mathbb{R}} \frac{1}{2}||\boldsymbol{w}||^2 + \frac{1}{\nu n}\sum_i^n \xi_i - \rho$$

subject to

$$\boldsymbol{w} \cdot \Phi(\boldsymbol{x_i}) \geq \rho - \xi_i,\ \xi_i \geq 0,\ \forall i = 1, \ldots, n.$$

We introduce slack variables $\xi_i \geq 0$ to let some points from the data out of description. In this way, the technique is robust to outliers and noisy data sets. The coefficient $\nu \in (0, 1)$

indicates the expecting fraction of outliers. Moreover, when ν tends to 0, we force every point in the database to be into the data description. This is the hard margin one-class SVM. Contrariwise, when we let some outliers out of description, we speak about soft margin one-class SVM.

We can show that the discriminating function f is defined by:

$$f(\boldsymbol{x}) = sign\left(\sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}) - \rho\right).$$

The coefficients $\boldsymbol{\alpha}$ can be found optimizing the corresponding dual problem

$$\min_{\boldsymbol{\alpha}} \frac{1}{2} \sum_{i=1,\, j=1}^{n} \alpha_i \alpha_j k(\boldsymbol{x_i}, \boldsymbol{x_j}),$$

subject to

$$0 \leq \alpha_i \leq \frac{1}{\nu n},\ \forall i \in \{1, \ldots, n\},\ \sum_{i=1}^{n} \alpha_i = 1.$$

The one-class SVM gives a normal data description of the data set and for a new point, we want to decide if it belongs to this normal description. If not, we can consider this point as an anomaly. More precisely, for any $\boldsymbol{x} \in \mathbb{R}^p$, $\boldsymbol{x}$ is considered as anomalous if $f(\boldsymbol{x}) = -1$, as normal otherwise.

We do not have shown how to derive the dual problem but it can be done, in the same way as for the SVDD. We refer the reader to the next section for more details.

2.3 Support Vector Data Description

The theory around SVDD has been developed based on the idea of the one-class SVM. In the same way, this technique allows the problem to be mapped into another feature space using kernel functions. The goal of the SVDD is to find a hypersphere containing all data points and minimizing its volume. However, due to noise in the data, the possibility to let outliers outside the hypersphere needs to be included into the problem. Then, slack variables are considered. SVDD has been first introduced by [41], but their formulation faces some optimization problems. First, the dual problem is not convex and might converge to a local optimum. Moreover, in [7] the authors have proved that the problem is infeasible for some value of the cost parameter C. These problems have been solved quickly in practice. However, we want to derive a rigorous theoretical formulation. This section is based on paper [8] proposing a theoretical point of view on this problem. Let Φ be the same mapping function as for the one-class SVM. We consider the data set $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^p$.

In [41], they proposed to solve the following optimization problem,

$$\begin{aligned} \min_{R\in\mathbb{R},\, \boldsymbol{a}\in F,\, \boldsymbol{\xi}\in\mathbb{R}^n} \; & R^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to } & ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 \leq R^2 + \xi_i,\ i = 1, ..., n, \\ & \xi_i \geq 0,\ i = 1, ..., n. \end{aligned} \tag{2.5}$$

The positive constant C controls the trade-off between the volume of the hypersphere and the number of points outside of it. However, this problem is not in the form of 2.1 since

$$||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 - R^2 - \xi_i$$

is concave with respect to R.

A convex reformulation of this problem is proposed in [8]:

$$\begin{aligned} \min_{\bar{R}\in\mathbb{R},\, \boldsymbol{a}\in F,\, \boldsymbol{\xi}\in\mathbb{R}^n} \; & \bar{R} + C\sum_{i=1}^{n}\xi_i \\ \text{subject to } & ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 \leq \bar{R} + \xi_i,\ i = 1, ..., n, \\ & \xi_i \geq 0,\ i = 1, ..., n, \\ & \bar{R} \geq 0, \end{aligned} \tag{2.6}$$

by defining $\bar{R} = R^2$. A constraint on the non-negativity of $\bar{R}$ has been added. Since

$$||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 - \bar{R} - \xi_i = \boldsymbol{a}^T\boldsymbol{a} - 2\Phi(\boldsymbol{x}_i)^T\boldsymbol{a} - \bar{R} - \xi_i + k(\boldsymbol{x_i}, \boldsymbol{x_i}),$$

where k is the kernel function related to Φ, is convex with respect to $\bar{R}$ and ξ_i, and is strictly convex with respect to $\boldsymbol{a}$, the problem 2.6 is in the form of 2.1. However, the objective function of 2.6 is not strictly convex. Then, this problem may accept more than one optimal solution. This is the purpose of the following theorem.

Theorem 2.3. *The problem 2.6 accepts an unique optimal value for $\boldsymbol{a}$, but the optimal values for $\bar{R}$ and $\boldsymbol{\xi}$ are not unique.*

Proof. This proof is done by contradiction. If $\boldsymbol{a}$ is not unique, then there exist at least two optimal solutions. Let $(\bar{R}_1, \boldsymbol{a}_1, \boldsymbol{\xi}_1)$ and $(\bar{R}_2, \boldsymbol{a}_2, \boldsymbol{\xi}_2)$ be these two solutions such that

$$\boldsymbol{a}_1 \neq \boldsymbol{a}_2 \tag{2.7}$$

and

$$\bar{R}_1 + C\sum_{i=1}^{n}(\boldsymbol{\xi}_1)_i = \bar{R}_2 + C\sum_{i=1}^{n}(\boldsymbol{\xi}_2)_i. \tag{2.8}$$

First, we want to show that the optimal objective value is positive. Otherwise, using the non-negativity constraints for $\bar{R}$ and $\boldsymbol{\xi}$ of problem 2.6 the objective value is equal to zero. This objective value is equal to zero if and only if we have

$$\bar{R}_1 = 0 = \bar{R}_2 \text{ and } \boldsymbol{\xi}_1 = \boldsymbol{0} = \boldsymbol{\xi}_2.$$

This implies that for all $i \in \{1, \dots, n\}$, we have

$$||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}_1||^2 = 0 = ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}_2||^2.$$

From this, we deduce

$$\boldsymbol{a}_1 = \boldsymbol{a}_2,$$

which is a contradiction to 2.7.

We define $d(\boldsymbol{a}) \equiv ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2$. This function is strictly convex with respect to $\boldsymbol{a}$. Since $\boldsymbol{a}_1 \neq \boldsymbol{a}_2$, there exists $\theta \in (0, 1)$ such that:

$$\begin{aligned} d(\theta\boldsymbol{a}_1 + (1-\theta)\boldsymbol{a}_2) &= ||\Phi(\boldsymbol{x}_i) - \theta\boldsymbol{a}_1 - (1-\theta)\boldsymbol{a}_1||^2 \\ &< \theta d(\boldsymbol{a}_1) + (1-\theta)d(\boldsymbol{a}_2) \\ &= \theta||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}_1||^2 + (1-\theta)||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}_2||^2 \\ &\leq \theta(\bar{R}_1 + (\boldsymbol{\xi}_1)_i) + (1-\theta)(\bar{R}_2 + (\boldsymbol{\xi}_2)_i), \end{aligned} \tag{2.9}$$

for all $i \in \{1, \dots, n\}$. We have found the last inequality using the constraints of 2.6.

From 2.9 we can deduce that there exists $\beta \in (0, 1)$ such that:

$$d(\theta\boldsymbol{a}_1 + (1-\theta)\boldsymbol{a}_2) \leq \beta(\theta(\bar{R}_1 + (\boldsymbol{\xi}_1)_i) + (1-\theta)(\bar{R}_2 + (\boldsymbol{\xi}_2)_i))$$

for all $i \in \{1, \dots, n\}$. It implies that

$$\theta(\beta\bar{R}_1, \boldsymbol{a}_1, \beta\boldsymbol{\xi}_1) + (1-\theta)(\beta\bar{R}_2, \boldsymbol{a}_2, \beta\boldsymbol{\xi}_2)$$

is a feasible point for 2.6. Moreover, the objective function value at this point is given by:

$$\beta\theta(\bar{R}_1 + C\sum_{i=1}^{n}(\boldsymbol{\xi}_1)_i) + \beta(1-\theta)(\bar{R}_2 + C\sum_{i=1}^{n}(\boldsymbol{\xi}_2)_i) = \beta(\bar{R}_1 + C\sum_{i=1}^{n}(\boldsymbol{\xi}_1)_i), \tag{2.10}$$

where the equality is true by 2.8. Since we have shown that the optimal objective value is positive, the value in 2.10 is strictly smaller than the objective value in 2.8 (since $\beta \in (0, 1)$), which is a contradiction. In conclusion, the optimal $\boldsymbol{a}$ is unique.

We give an example to show that the optimal $\bar{R}$ and $\boldsymbol{\xi}$ are not unique. Consider the problem 2.6 with $C = 1/2$, Φ the identity function and $\boldsymbol{x}_1 = 1$, $\boldsymbol{x}_2 = -1$ the two input objects.

We have, for all feasible $\boldsymbol{a}$

$$\bar{R} + C(\xi_1 + \xi_2) = \frac{1}{2}((\bar{R} + \xi_1) + (\bar{R} + \xi_2)) \geq \frac{1}{2}((\boldsymbol{a} - 1)^2 + (\boldsymbol{a} + 1)^2). \tag{2.11}$$

Consider two points $(\bar{R}_1, a, \boldsymbol{\xi}_1)$ and $(\bar{R}_2, a, \boldsymbol{\xi}_2)$ defined with:

$$\begin{aligned} \bar{R}_1 = 1,\ \boldsymbol{a} = 0,\ \boldsymbol{\xi}_1 = (0, 0), \\ \bar{R}_2 = 1/2,\ \boldsymbol{a} = 0,\ \boldsymbol{\xi}_2 = (1/2, 1/2). \end{aligned} \tag{2.12}$$

Clearly, they are both feasible. Moreover, it shows that $\mathbf{0}$ is a feasible value for $\boldsymbol{a}$. In particular, since 2.11 is true for all feasible $\boldsymbol{a}$, we have:

$$\bar{R} + C(\xi_1 + \xi_2) \geq \frac{1}{2} \min_{\boldsymbol{a}:\boldsymbol{a} feasible} (\boldsymbol{a} - 1)^2 + (\boldsymbol{a} + 1)^2 = 1.$$

We conclude that the optimal objective value is at least 1. Furthermore, the objective value of $(\bar{R}_1, a, \boldsymbol{\xi}_1)$ and $(\bar{R}_2, a, \boldsymbol{\xi}_2)$ is 1. Then, they are both optimal. □

Theorem 2.4. *For the convex optimization problem 2.6 the strong duality holds.*

Proof. This is a corollary of the theorem 2.2.

The problem 2.6 does not have equality constraints. Moreover, the domains of $f, g_1, \ldots, g_{2n+1}$ are the same Euclidean space. It implies that any point in this Euclidean space belongs to its *relint*.

Let $\boldsymbol{a} = \mathbf{0}$, $\bar{R} = 1$, and

$$\xi_i = ||\Phi(\boldsymbol{x}_i)||^2 + 1,\ i = 1, ..., n.$$

Trivially $(\bar{R}, \boldsymbol{a}, \boldsymbol{\xi})$ is a feasible solution and

$$\begin{aligned} -\bar{R} &= -1 < 0, \\ -\xi_i &\leq -1 < 0, \\ ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 - \bar{R} - \xi_i &= -2 < 0,\ i = 1, \ldots, n. \end{aligned}$$

The point $(\bar{R}, \boldsymbol{a}, \boldsymbol{\xi})$ satisfies the theorem 2.2 and therefore the strong duality holds. □

In conclusion, we can solve the problem 2.6 by its dual. However, the non-negativity constraint for $\bar{R}$ makes the derivation of the dual more difficult. The following theorem will allow us to simplify the dual.

Theorem 2.5. *If we consider the problem 2.6, the following is true.*

- *For any $C > 1/n$, the constraint $\bar{R} \geq 0$ is not necessary. Moreover, without this constraint, any optimal solution still satisfies $\bar{R} \geq 0$.*
- *For any $0 < C < 1/n$, $\bar{R} = 0$ is uniquely optimal. If $C = 1/n$, then at least one optimal solution accepts $\bar{R} = 0$.*

Proof.

- Consider the problem 2.6 without the constraint $\bar{R} \geq 0$ and with $C > 1/n$.

  This proof is done by contradiction. Assume that the problem has an optimal point $(\bar{R}, \boldsymbol{a}, \boldsymbol{\xi})$ with $\bar{R} < 0$. We claim that $(0, \boldsymbol{a}, \boldsymbol{\xi} + \bar{R}\boldsymbol{e})$ is a feasible point, where $\boldsymbol{e}$ is the unit vector of ones. Indeed, since $(\bar{R}, \boldsymbol{a}, \boldsymbol{\xi})$ is optimal and thus feasible, we have

  $$0 \leq ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 \leq \bar{R} + \xi_i = 0 + (\xi_i + \bar{R}),\ i = 1, \ldots, n,$$

and furthermore,

$$\xi_i + \bar{R} \geq 0,\ i = 1, \ldots, n.$$

Moreover, the objective value for $(0, \boldsymbol{a}, \boldsymbol{\xi} + \bar{R}\boldsymbol{e})$ satisfies

$$0 + C\sum_{i=1}^{n}(\xi_i + \bar{R}) = C\sum_{i=1}^{n}\xi_i + Cn\bar{R} < C\sum_{i=1}^{n}\xi_i + \bar{R},$$

where we find the inequality because $C > 1/n$ and $\bar{R} < 0$. This is a contradiction on the optimality of $(\bar{R}, \boldsymbol{a}, \boldsymbol{\xi})$. We conclude that any feasible solution for 2.6 (without the constraint on the non-negativity of $\bar{R}$) with $\bar{R} < 0$ is never optimal when $C > 1/n$.

- Consider the problem 2.6 with $C \leq 1/n$. Again, we proceed by contradiction. We assume that $(\bar{R}, \boldsymbol{a}, \boldsymbol{\xi})$ is an optimal point with $\bar{R} > 0$. Then, the point $(0, \boldsymbol{a}, \boldsymbol{\xi} + \bar{R}\boldsymbol{e})$ is feasible since

$$0 \leq ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 \leq \bar{R} + \xi_i = 0 + (\xi_i + \bar{R}),\ i = 1, \ldots, n,$$

and furthermore,

$$\xi_i + \bar{R} \geq 0,\ i = 1, \ldots, n.$$

Moreover, the objective value for $(0, \boldsymbol{a}, \boldsymbol{\xi} + \bar{R}\boldsymbol{e})$ satisfies

$$0 + C\sum_{i=1}^{n}(\xi_i + \bar{R}) = C\sum_{i=1}^{n}\xi_i + Cn\bar{R} \leq C\sum_{i=1}^{n}\xi_i + \bar{R}, \tag{2.13}$$

where, we find the inequality because $C \leq 1/n$ and $\bar{R} > 0$. We deduce that the new point with $\bar{R} = 0$ is optimal. Then, there always exists an optimal solution with $\bar{R} = 0$ when $C \leq 1/n$. Furthermore, when $C < 1/n$ the inequality of 2.13 is strict and then we find a contradiction on the optimality of $(\bar{R}, \boldsymbol{a}, \boldsymbol{\xi})$. It proves that any feasible solution with $\bar{R} > 0$ is not optimal for 2.6 when $C < 1/n$. Then, the unique optimal solution has $\bar{R} = 0$.

□

Since the goal of this master thesis is to detect anomalies using support vector data description, the problem corresponding to an optimal solution with $\bar{R} = 0$ is not interesting for us. In the following, we will only consider the case where $C > 1/n$. Now, we will derive the Lagrangian of the modified problem:

$$\begin{aligned} \min_{\bar{R}\in\mathbb{R},\, \boldsymbol{a}\in F,\, \boldsymbol{\xi}\in\mathbb{R}^n} \quad & \bar{R} + C\sum_{i=1}^{n}\xi_i \\ \text{subject to } & ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 \leq \bar{R} + \xi_i,\ i = 1, ..., n, \\ & \xi_i \geq 0,\ i = 1, ..., n, \end{aligned} \tag{2.14}$$

where $C > 1/n$. Given this problem, the Lagrangian is:

$$\begin{aligned}\mathcal{L}(\bar{R}, \boldsymbol{a}, \xi_i, \boldsymbol{\alpha}, \boldsymbol{\mu}) &= \bar{R} + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i\left(\bar{R} + \xi_i - (k(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2\boldsymbol{a}^T\Phi(\boldsymbol{x}_i) + \boldsymbol{a}^T\boldsymbol{a})\right) - \sum_{i=1}^{n}\mu_i\xi_i \\ &= \bar{R}\left(1 - \sum_{i=1}^{n}\alpha_i\right) + \sum_{i=1}^{n}\xi_i(C - \alpha_i - \mu_i) + \sum_{i=1}^{n}\alpha_i(k(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2\boldsymbol{a}^T\Phi(\boldsymbol{x}_i) + \boldsymbol{a}^T\boldsymbol{a}),\end{aligned} \tag{2.15}$$

where $\boldsymbol{\alpha} \geq 0$ and $\boldsymbol{\mu} \geq 0$ are the Lagrange multipliers. The Lagrange dual problem needs to be minimized with respect to $\bar{R}$, $\boldsymbol{a}$, ξ_i and maximized with respect to $\boldsymbol{\alpha}, \boldsymbol{\mu}$. Deriving the Lagrangian with respect to $\bar{R}$, $\boldsymbol{a}$ and $\boldsymbol{\xi}$ and equalizing these expressions to 0, we get the following constraints:

$$\frac{\partial\mathcal{L}(\bar{R}, \boldsymbol{a}, \xi_i, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial\bar{R}} = 0 \Rightarrow \sum_{i=1}^{n}\alpha_i = 1, \tag{2.16}$$

$$\frac{\partial\mathcal{L}(\bar{R}, \boldsymbol{a}, \xi_i, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial\boldsymbol{a}} = 0 \Rightarrow \boldsymbol{a} = \frac{\sum_{i=1}^{n}\alpha_i\Phi(\boldsymbol{x}_i)}{\sum_{i=1}^{n}\alpha_i} = \sum_{i=1}^{n}\alpha_i\Phi(\boldsymbol{x}_i), \tag{2.17}$$

$$\frac{\partial\mathcal{L}(\bar{R}, \boldsymbol{a}, \xi_i, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial\xi_i} = 0 \Rightarrow \alpha_i = C - \mu_i,\ i = 1, \ldots, n. \tag{2.18}$$

Using equation (2.18) we can rewrite the positivity constraints on the Lagrange multipliers with the constraints

$$0 \leq \alpha_i \leq C,\ i = 1, \ldots, n.$$

By replacing 2.16, 2.17 and 2.18 into the Lagrangian 2.15, we find the dual problem:

$$\begin{aligned}&\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n}\alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_i) - \sum_{i=1,j=1}^{n}\alpha_i\alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ &\text{subject to } 0 \leq \alpha_i \leq C,\ i = 1, ...n, \\ &\sum_{i=1}^{n}\alpha_i = 1.\end{aligned} \tag{2.19}$$

Many easy calculus have been omitted and the reader is free to do them.

When we have found an optimal solution for the dual, we can easily deduce the optimal solution for $\boldsymbol{a}$ using 2.17. However, finding an optimal solution for $\bar{R}$ is not so easy. Since this problem satisfies the strong duality, the KKT conditions are necessary for optimal solutions. Therefore, primal and dual optimal solutions satisfy the two complementary slackness conditions:

$$\alpha_i(\bar{R} + \xi_i - ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2) = 0, \tag{2.20}$$

$$\text{and } \mu_i\xi_i = 0,\ i = 1, \ldots, n. \tag{2.21}$$

If we consider 2.18 along with 2.20, for all i such that $0 < \alpha_i < C$ we have

$$\xi_i = 0, \tag{2.22}$$

$$||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 = \bar{R} + \xi_i, \tag{2.23}$$

and we can find an optimal solution for $\bar{R}$. However, if such and i does not exists, it is not so easy to find an optimal $\bar{R}$; the following theorem answers to this problem.

Theorem 2.6. *When $C > 1/n$, given the optimal $\boldsymbol{a}$ of 2.14 and an optimal $\boldsymbol{\alpha}$ of 2.19, a feasible $\bar{R}$ is optimal for 2.14 if and only if*

$$\max_{i:\, \alpha_i < C} ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 \leq \bar{R} \leq \min_{i:\, \alpha_i > 0} ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2. \tag{2.24}$$

Proof. From 2.20, 2.18 and the constraints of 2.14, we reap

$$\begin{aligned} \bar{R} &\geq ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2, \ \forall i : \alpha_i < C, \\ \bar{R} &\leq ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2, \ \forall i : \alpha_i > 0, \end{aligned} \tag{2.25}$$

which conclude the proof. The "if and only if" is true because the KKT conditions are necessary and sufficient conditions for optimality when the Slater's condition holds (see 2.2) and the problem is convex. For more details about how the conditions 2.25 are found, see the Remark 2.1 below. □

The last theorem shows that if there exists an index i with $0 < \alpha_i < C$, 2.24 is simplified to 2.22 and 2.23, then the optimal $\bar{R}$ is unique. Contrariwise, if every α_i is either equal to 0 or C, thus every $\bar{R}$ in the interval described by 2.24 is optimal. In practice we compute $\bar{R}$ using the two following rules:

- If some indices satisfy $0 < \alpha_i < C$, then we calculate $\bar{R}$ as the average of $||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2$ over all such i. We take the average value to avoid some numerical errors.
- If for all indices i, $\alpha_i = C$ or $\alpha_i = 0$, then we take $\bar{R}$ as the lower bound of the interval described in 2.24. The reason is that we want to minimize the volume of the hypersphere.

Moreover, we can compute the optimal value of $\boldsymbol{\xi}$ from the primal function 2.14 by

$$\xi_i = \max(||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 - \bar{R}, 0), \ i = 1, \ldots, n. \tag{2.26}$$

When the value of C is large enough, the primal problem 2.14 can be further simplified. This is the point of the following theorem.

Theorem 2.7. *For any $C > 1$, the problem 2.14 can be simplified to the following problem:*

$$\begin{aligned} &\min_{\bar{R} \in \mathbb{R},\, \boldsymbol{a} \in F} \bar{R} \\ &\textit{subject to } ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 \leq \bar{R} \end{aligned} \tag{2.27}$$

Proof. Since $\alpha_i \geq 0$ for all i, along with 2.16, 2.18 and if $C > 1$, we have $\mu_i > 0$ for all $i \in \{1, \ldots, n\}$.

Then, using 2.20 for an optimal solution we have $\xi_i = 0$, for all $i \in \{1, \ldots, n\}$. Thus, the term $C \sum_{i=1}^{n} \xi_i$ can be removed from the objective function. This is sufficient to conclude the proof. □

The difference between the formulation of the problems 2.14 and 2.27 is the same as the one of the soft and hard margin formulations for the one-class SVM.

Remark 2.1. Finally, we can hold a discussion on the value of α_i, $i = 1, \ldots, n$.

- If $\alpha_i = 0$, μ_i must be equal to C. Thus, ξ_i must be equal to 0 and the point $\boldsymbol{x}_i$ lies inside of data description (including the contour).
- If $0 < \alpha_i < C$, $||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2$ must be equal to $\bar{R}$, and thus the point $\boldsymbol{x}_i$ lies on the contour.
- If $\alpha_i = C$, $||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2$ must be equal to $\bar{R} + \xi_i$; then, the point $\boldsymbol{x}_i$ lies outside of the data description (including the contour) since ξ_i is non-negative.

The *Support Vectors* are the points for which $\alpha_i > 0$. Only the *support vectors* are needed to compute the value of $\bar{R}$ because for any index i using 2.17, we have

$$\begin{aligned} ||\Phi(\boldsymbol{x}_i) - \boldsymbol{a}||^2 &= k(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2\boldsymbol{a}^T \Phi(\boldsymbol{x}_i) + \boldsymbol{a}^T \boldsymbol{a} \\ &= k(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2 \sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{j=1,\, k=1}^{n} \alpha_j \alpha_i k(\boldsymbol{x}_j, \boldsymbol{x}_i) \\ &= k(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2 \sum_{j:\, \alpha_j > 0} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \boldsymbol{\alpha}^T \boldsymbol{G} \boldsymbol{\alpha}, \end{aligned}$$

where $\boldsymbol{G}$ is the Gramian matrix associated with the kernel function k and the data points. The element i, j of the matrix $\boldsymbol{G}$ is defined by $(\boldsymbol{G})_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, for all $i, j = 1, \ldots, n$.

For a new data instance $\boldsymbol{x} \in \mathbb{R}^p$, we want to test if it lies inside or outside the data description. This is done by looking at the following expression

$$||\Phi(\boldsymbol{x}) - \boldsymbol{a}||^2 - \bar{R} = k(\boldsymbol{x}, \boldsymbol{x}) - 2 \sum_{i:\alpha_i > 0} \alpha_j k(\boldsymbol{x}, \boldsymbol{x}_i) + \boldsymbol{\alpha}^T \boldsymbol{G} \boldsymbol{\alpha} - \bar{R}. \tag{2.28}$$

If 2.28 is positive, the point $\boldsymbol{x}$ lies outside of data description. Otherwise, $\boldsymbol{x}$ lies inside the data description (including the contour). For kernels $k(\boldsymbol{x}, \boldsymbol{y})$ which only depend on $\boldsymbol{x} - \boldsymbol{y}$, $k(\boldsymbol{x}, \boldsymbol{x})$ is constant. In this case, we can show that the problem 2.14 is equivalent to the one-class SVM (see [36]). The positive constant C can be tuned in order to control the number of points from the data to let outside of the data description. Indeed, for any index $i = 1, \ldots, n$ a necessary condition for $\boldsymbol{x}_i$ to be outside of the description is that the

corresponding α_i is equal to C, such a point is called an *error*. Using the condition 2.16, we have

$$\#errors \leq \frac{1}{C}.$$

In particular, when $C > 1$ the number of errors is 0, which provides an alternative proof to 2.7. Let ν be the expected fraction of outliers in the data, we should choose $C \leq \frac{1}{n\nu}$.

2.3.1 Improvements and similar methods

Many other methods exist in the data description problem. As said in the previous sections, the goal is to find if a new data object is similar in distribution to a learning sample. For detecting outliers, an advanced Bayesian approach is used for example in paper [4]. In this technique, the output is weighted by the probability that the weight configuration is correct. This method provides an estimate of the probability for a certain object to belong to the distribution of the data. Then, a low probability associated with a output will possibly mean that this object is an outlier.

There are many cases in the literature where the data description is done by estimating a probability density of the learning data. For example, in [3] the density is estimated by a Parzen density estimator, while in [28] a Gaussian distribution is used. The drawbacks of these methods are that in a high dimensional case, a lot of training samples are required. Moreover, these methods are not very robust to training data, while SVDD is resistant to changes in training data which are not support vectors.

In [29], the original SVDD problem is revisited. They focus on the primal form of the problem, instead of its dual and try to give a geometric interpretation. Moreover, they look at an alternative problem considering the non-squared distance and found that in some situations this method is preferable to the original. In fact, these two methods differ in the behavior in almost the same way as the mean and the median.

The main drawback of SVDD is its computational cost. Many literature papers have tried to solve this problem by slightly modifying the algorithm. For instance, [18] proposes an incremental learning to train the model. Instead of using all the available data to find the best data description they proceed by iteration. They begin by a small subset of the data and at each iteration they increase the number of training samples. Then, the new solution is not independent from the previous data description and can be computed a lot faster. They barely reach the same performance as for a classic SVDD with a reduced computation time. In [23], a method called Fast SVDD replacing the kernel expansion of the SVDD by a single kernel term is presented. It relies on the computation of a single pre-image of a point called the *agent of the SVDD sphere's center*. In [32], another solution inspired by the Fast SVDD called the Lightly trained SVDD is described. The training phase of this algorithm learns a pre-image (as for the Fast SVDD) in the input space instead of computing the Lagrange multipliers α_i. In order to do that, they use the primal problem formulation proposed in [29]. The result approximates the solution of the real

SVDD. Both fast SVDD and lightly trained SVDD, are faster than the classic SVDD and perform as well as SVDD. However, for some database the use of the Gaussian Kernel is needed. In this case, the authors have found that the decision hypersphere become a sphere in the input space and then, this kernel is useless.

Chapter 3

LSTM and SVDD for anomaly detection

In this chapter, we will explain how we combine both LSTM and SVDD for anomaly detection. Afterward, we will show how to build an LSTM architecture in practice, including a discussion on the choice of the many parameters to be optimized.

We will use two LSTM-base techniques to detect anomaly using reconstruction errors. Given its reconstruction error, a point will be detected as anomalous if this error does not lie inside the data description.

In [24], they used a LSTM-based Auto-Encoder-Decoder to obtain the reconstruction errors and discriminate these reconstruction errors by computing their likelihoods under normality assumption. Similarly, in [25], they assumed normality of the non-anomalous reconstruction errors but a used different LSTM architecture. In this master thesis, we do not assume normality of the errors and propose a more flexible data description using SVDD with kernels.

In [11], the author proposed to combine both LSTM and SVDD for anomaly detection. However, LSTM and SVDD are jointly trained with a newly created algorithm, which is out of scope for this master thesis.

3.1 LSTM-based anomaly detection

This section is mainly based on the paper [25].

Consider a time series $X = \{\boldsymbol{x}(1), \boldsymbol{x}(2), \ldots, \boldsymbol{x}(T)\}$ where each point of this series is a p-dimensional vector $(x_1(t), \ldots, x_p(t))^T$, $t = 1, \ldots, T$. Consider the associated event time series $Y = \{y(1), y(2), \ldots, y(T)\}$ where each point of this time series is either 1 or 0. If $y(t) = 1$, then the associated point $\boldsymbol{x}(t)$ is considered as anomalous, otherwise, $\boldsymbol{x}(t)$ is considered as normal, $t = 1, \ldots, T$.

We build a LSTM-prediction model which uses the l previous values as input variables to predict the next m values as output variables.

More precisely, given $\{\boldsymbol{x}(t-l+1), \boldsymbol{x}(t-l+2), \ldots, \boldsymbol{x}(t)\}$ the model tries to predict $\{\boldsymbol{x}(t+1), \ldots, \boldsymbol{x}(t+m)\}$ with $l \geq 1$, $m \geq 1$, $t = l-1, \ldots, T-m$. When $m > 1$ we need to use a LSTM Encoder-Decoder architecture (see Figure 3.1). The goal of the Encoder-Decoder architecture is to adapt the number of time steps of the network's outputs to the considered target (mathematical details are give in the appendix C).

To train the LSTM model we need to divide the whole time series X into sub-time series. Let $X_t \equiv \{\boldsymbol{x}(t-l+1), \boldsymbol{x}(t-l+2), \ldots, \boldsymbol{x}(t), \boldsymbol{x}(t+1), \ldots, \boldsymbol{x}(t+m)\}$, $l \geq 1$, $m \geq 1$, $t = l-1, \ldots, T-m$, be one such sub-time series. We define a sub-time series as anomalous if it contains at least one anomalous point and as normal otherwise. After dividing the time series X with fixed values for l and m, we have a set of sub-time series $\{X_{l-1}, \ldots, X_{T-m}\}$ associated with a set of labels.

With a prediction length of m, each of the p dimensions of $\boldsymbol{x}(t) \in X$, $l-1 \leq t \leq T-m$, is predicted m times. We compute the error vector $\boldsymbol{e}(t)$ for the point $\boldsymbol{x}(t)$ as $\boldsymbol{e}(t) = (e(t)_{11}, \ldots, e(t)_{1m}, \ldots, e(t)_{p1}, \ldots, e(t)_{pm})$, where $e(t)_{ij}$ is the difference between $x_i(t)$ and its predicted value at time $t-j$.

In this master thesis, we will consider two ways of detecting an anomaly given an error vector, and we will compare them. Firstly, we will construct a normal error data description using SVDD and a point will be detected as anomalous if it does not belong to this description. Secondly, we will suppose that the normal errors follow a gaussian distribution and a point will be detected as anomalous if its anomaly score is above a certain threshold.

In the first case, we construct a normal data description for the errors by optimizing a SVDD on a set of errors coming from normal points. When we assume the Gaussian distribution of the error vectors, we estimate the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ of the normal distribution $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ using Maximum Likelihood Estimation on a set of normal error vectors. For a new point $\boldsymbol{x}(t)$ we compute its anomaly score $a(t)$ as $a(t) = (\boldsymbol{e}(t) - \boldsymbol{\mu})^T \boldsymbol{\Sigma} (\boldsymbol{e}(t) - \boldsymbol{\mu})$. If $a(t) > \tau$, the point is considered as anomalous, otherwise as normal. The threshold τ is chosen to maximize the performance on a validation set. More details will be given in Chapter 4.

3.2 LSTM-based Auto-Encoder-Decoder anomaly detection

This section is mainly based on the article [24]. The same notations as in the previous section are used.

Consider a scenario where multiple time series $Z = \{\boldsymbol{z}(1), \boldsymbol{z}(2), \ldots, \boldsymbol{z}(L)\}$ are available with $L \geq 1$. From the time series X defined in the previous section we can simply extract

$\lfloor T/L \rfloor$ such disjoint time series.

Given Z the LSTM is trained to reconstruct the time series in reverse order as proposed in [39], i.e. the target time-series is $\{\boldsymbol{z}(L), \boldsymbol{z}(L-1), \ldots, \boldsymbol{z}(1)\}$.

It seems to be a trivial task since the inputs are perfectly correlated with the targets. Then, we use an Auto-Encoder-Decoder architecture. The role of the encoder is to merge the input sequence into a single vector, called the *encoded representation*, and keep only the useful information in the model. Contrariwise, the role of the decoder is to reconstruct the original sequence from the encoded representation. For a graphical intuitive representation of an (Auto-)Encoder-Decoder architecture see Figure 3.1. Details on how an Encoder-Decoder LSTM is defined mathematically are given in the appendix C.

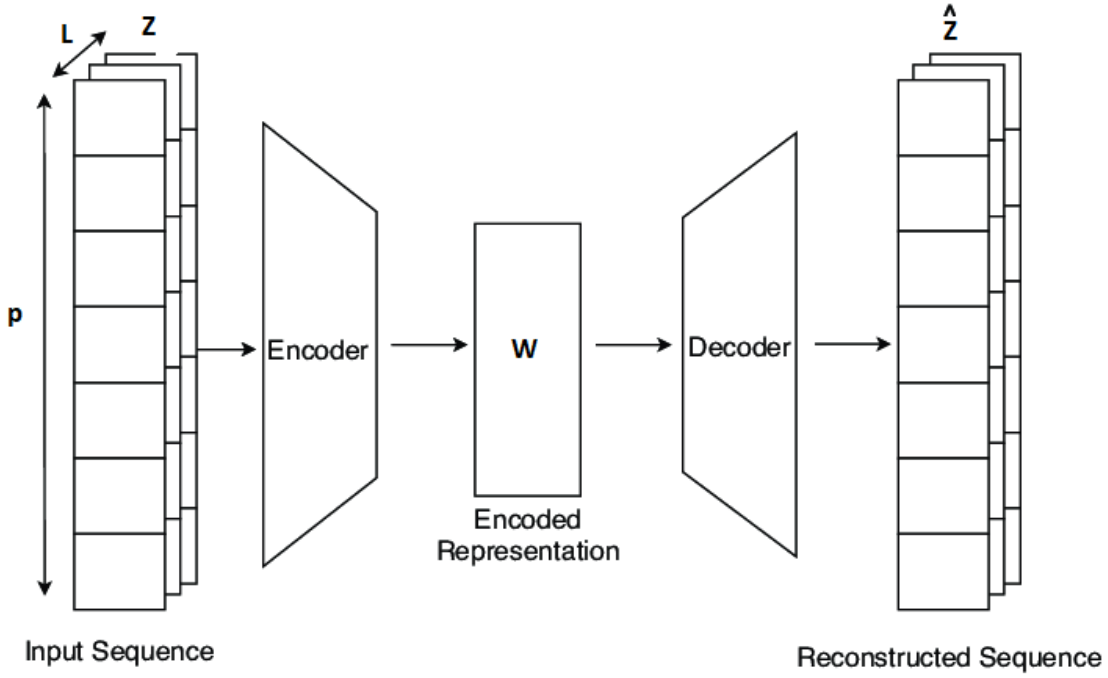


Figure 3.1 – *Representation of an Encoder-Decoder architecture where* $\boldsymbol{w}$ *is the encoded representation vector.*

For a given time series $Z = \{\boldsymbol{z}(1), \boldsymbol{z}(2), \ldots, \boldsymbol{z}(L)\}$, we define the reconstruction error vector $\boldsymbol{e}(i)$ for any point of the time series $\boldsymbol{z}(i)$, $i = 1, \ldots, L$, by

$$e(i)_j = z(i)_j - \hat{z}(i)_j,$$

$j = 1, \ldots, p$, where $\hat{\boldsymbol{z}}(i)$ is the predicted vector of $\boldsymbol{z}(i)$ through the network.

If the Encoder-Decoder model is trained on the normal time series, a too large (with respect to our data description) reconstruction error for any point will lead to detect it as an anomaly as in the previous section.

3.3 LSTM in practice

We consider the software R to apply the LSTM in practice. We use the R package keras, which is a python interface, to build neural networks. Creating a neural network architecture with keras is done by stacking layers into a sequential keras model. Then, we can deeply customize our architecture. In this section, we will detail the main parameters of a LSTM neural network and how to choose them in order to tune the network. However, the building of neural networks is far from hard science and there do not exist mathematical rules to choose these parameters. Thus, the hints that we will give are not general and need to be adapted with respect to the task, the data set and the goal to achieve.

When we optimize the architecture and the parameters of a neural network, we need to reach the best performance without overfitting the training data. The definition of the overfitting is "*the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably*" OxfordDictionaries.com[1]. Thus, a too complex neural network performs very well on the training set and reaches a good performance but may perform poorly on another unseen set (coming from the same data). Indeed, it risks to model not only the data distribution, but also the errors which is unwanted. In contrary, a too simple model does not perform well enough on both training and unseen sets.

First, we will present what are the parameters to tune and how to do it, we will speak about the building and the training of the network. Afterward, we will present a way to prevent over-fitting, the regularization. Finally, we will develop another architecture of LSTM, the bidirectional LSTM. This architecture allows the network to process the sequence in reverse order.

3.3.1 Building of the network

To define the topology of a LSTM neural network we need to tune two main parameters: the number of hidden layers and of hidden units. The more hidden layers and units a network is composed of, the more complex it is.

Even for difficult tasks it has been shown that a two hidden layers deep LSTM neural network is complex enough to reach good performance (see for example [34]). More layers can be better but also harder to train. Thus, we will not consider more than two hidden layers.

To choose the best number of units for the considered layers, we proceed by k-fold cross validation on the training set. We define a grid of values, and the model with the best performance is selected. Moreover, we generally choose the number of hidden units in the first hidden layer smaller or equal to the number of input variables to not create useless collinearity in the network variables.

[1] Available via the URL https://www.lexico.com/definition/overfitting. Accessed 04/29/2020.

With respect to the task we need also to choose the activation functions for the hidden layers and the output layer. In this master thesis we choose to consider the activation functions used in the vanilla LSTM in keras.

We use a hyperbolic tangent function for the input/ouput/forget gates activation and a hard sigmoid is used for the cell and the hidden state. The hard sigmoid is defined as

$$hs(x) \equiv \begin{cases} 0 & x < -2.5 \\ 0.2x + 0.5 & -2.5 \leq x \leq 2.5 \\ 1 & x > 2.5 \end{cases}$$

For a comparison between the *hard sigmoid*, the *sigmoid* and the tanh functions see Figure 3.2.

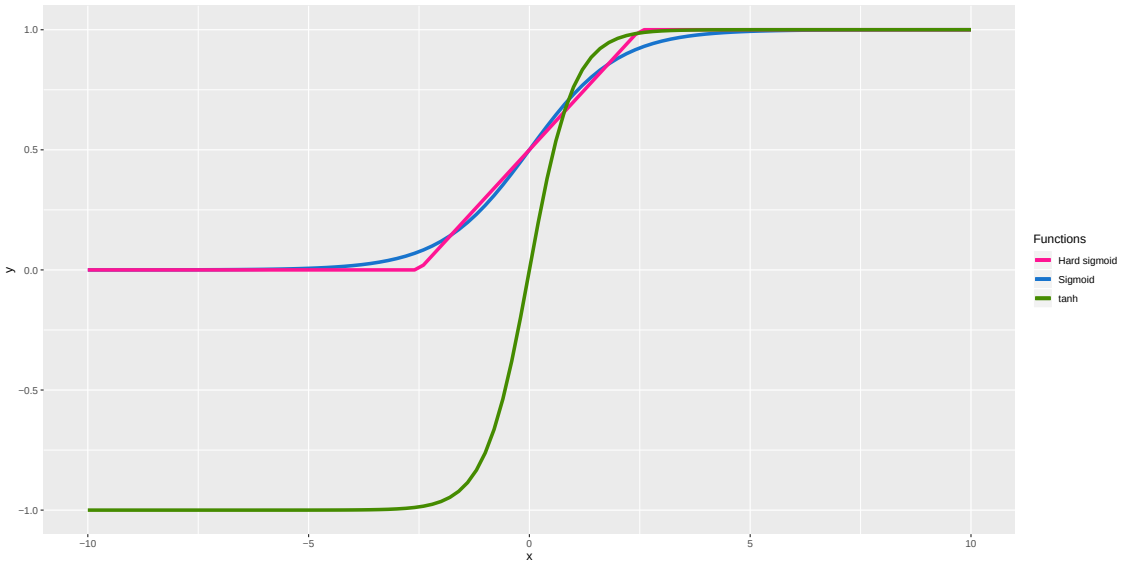


Figure 3.2 – *Comparison of different activation functions.*

We see from the figure that the *hard sigmoid* approximates the *sigmoid* function; the former is preferred over the latter for computational speed reasons. Mathematically, we will always consider $f_{in} = f_{out} = f_{\varphi} = \tanh$ and $h = g = hs$. Finally, for the dense output layer we simply choose the identity as activation function.

3.3.2 Training of the network

We will divide the section about the training of the network into four sub-sections, namely the loss function, the optimizer, the batch size and the number of epochs.

Loss function

The first step to train a neural network is to numerically compute the gradient of the loss function with respect to the weights; thus, we need to choose a loss function. Since our goal is to minimize the error vectors for the prediction on normal time series, we will consider the *mean squared error*. For a k-dimensional vector of outputs this error is defined by

$$E = \frac{\sum_{i=1}^{k}(\hat{y}_i - y_i)^2}{k},$$

where $\boldsymbol{y}$ is the target and $\hat{\boldsymbol{y}}$ its predicted value.

Optimizer

In the section 1.1.4 we have presented an optimizer to update the weights once the gradient is computed, the gradient descent. However, this optimizer is not used in practice. The reason is that if our loss function is not strictly convex with respect to the weights the probability to get stuck in a saddle point which is not a global optimum is high.

A lot of more developed optimization algorithms have been presented through years, and it is not always easy to know which is the most adapted algorithm to our problem. For example, a slightly modified gradient descent with momentum is still one of the best performing optimizers. Intuitively, the momentum helps the network to outpass the local minimum. However, its computational cost is expensive and it converges slowly according to [33]. Thus, we prefer a newer and faster optimizer using the adaptive moment estimator, called the Adam optimizer.

In [19], the authors have shown that the Adam optimizer slightly outperforms the other optimizers in convex and non-convex machine learning tasks. Moreover, Adam has been created to be able to deal with high dimensional space data set. We will also use and presents two variants of this optimizer, the Adamax and the Nadam algorithms. A mathematical description of these three algorithms is available in the appendix A.

Batch size

When we have selected the loss function and the optimizer, we need to choose the update frequency which we have called batch size. We have described what is the batch size in section 1.1.4. However, we have not discussed how we select this parameter. First, we will compare stochastic and batch update. Afterward, we will discuss about a mini-batch alternative.

In the late 1990s/early 2000s the literature around machine learning was torn. Some authors said that batch learning is faster, others said that online learning is better because it follows more closely the real gradient of the loss function.

Nevertheless, in [46] the authors have attempted to close this debate. They have proved and explained empirically why online training is in fact as fast as batch training and more accurate. It cannot be considered that this paper contains the absolute truth, but it presents solid arguments and results.

Batch learning consists in updating the weights after one complete epoch, geometrically it can be seen as computing the gradient for all individuals considering the initial weights and "sticking" together the resulting vectors. The problem of this technique is that the gradient with respect to the initial weights is always considered. In other words, the gradient determines where the weights should go from their current position but with respect to the initial weights, even if the gradient may have changed considerably between those two points. This technique may also lead to weights that overshoot the minimum and never come back to it. The descent actually follows a straight line and the only way to control this straight line is to reduce its length and then the learning rate.

On the other hand, considering online learning the gradient is computed over the current weights and then allows the gradient to follow the curve. This gradient can fluctuate around the true gradient since the computation is done for each individual and can overshoot a minimum or move into a wrong direction due to noise of the individuals. However, the correction can be done in the very next instance, whereas for batch learning a full epoch needs to be run in order to do this correction and then follows a wrong direction until the next epoch.

Since the learning rate for the batch size needs to be reduced it takes more time to converge to the minimum and then the online learning is faster.

The empirical results of [46] will not be displayed here but these results confirm the theoretical expectations. The results showed that online training needs a lot less epochs in order to reach their maximum accuracy than batch learning (up to 100 times faster computing time) when using the best learning rates. Therefore, the two different maximum accuracies are close and then, even when batch learning reaches a better accuracy, the gain does not worth to increase the computing time.

Both [46] and [27] have discussed about the best batch size to use in mini-batch alternative in order to reach a good performance in the fastest time. Their conclusions are similar. As the batch size decreases the accuracy of the training increases down to batch size of 2 or 4. Moreover, the best results have been obtained with batch sizes always smaller than 32. In [46], they have found empirically that a batch size smaller than 10 achieved barely the same performance as online learning for both accuracy and training time.

Number of epochs

An epoch is completed when the full training set has been seen by the network during the training procedure. Since the training of a network is not deterministic, training the network for more than one epoch is sometimes valuable. The more epochs we train the network on, the more the network tends to over-fit the training data. Generally, we want

to stop the training just before it begins to overfit. This can be done with early stopping. We provide a validation set to the network. The network is train only on the training set, but its performance is evaluated after each epoch on the validation set. When the performance of the network begins to decrease on the validation set, we stop the training procedure. Early stopping is generally combined with regularization techniques to prevent overfitting.

3.3.3 Regularization

In this section, we will present one of the most important point in the building of a neural network, the regularization. We will discuss about the 3 most used techniques of regularization: L1 and L2 regularization and dropout.

We can also prevent overfitting by increasing the number of training samples. Indeed, the more data is available, the more it is difficult for the network to model the errors of the full training set. However, the size of the data set is generally fixed by the problem and can be limited in real world applications.

L1 and L2 regularization

The philosophy of L1 and L2 regularization is the same, we penalize too large weights. We let a weight to be big if and only if the decrease of the loss function value is important and compensates the penalization. More precisely, let E be the considered loss function, when we add a L2 penalization term we consider a new error (or loss) function C defined by

$$L = E + \frac{\lambda}{2n} \sum_w w^2,$$

where $\lambda > 0$ is the regularization parameter, n the training set size and w holds for any weight in the network. Similarly, when we use a L1 penalization we consider a new cost function

$$L = E + \frac{\lambda}{2n} \sum_w |w|.$$

L1 regularization leads to a sparse optimization for the weight. However, sparse weights do not mean that the model can be simplified. Indeed, to remove a unit and then simplify the network we need that all ongoing weights from that unit are equal to zero, which is not often the case. In [35], they proposed a new type of L1 regularization leading to a group sparsity, but this subject is out of scope for this master thesis.

We can also imagine a L1-L2 regularization where we simultaneously penalize the L1-norm and the L2-norm of the weights. But why penalizing too large weights will prevent overfitting? The explanation is quite intuitive: suppose that our network mostly has small weights, which is expected with L1 and L2 penalization. The smallness of the weights

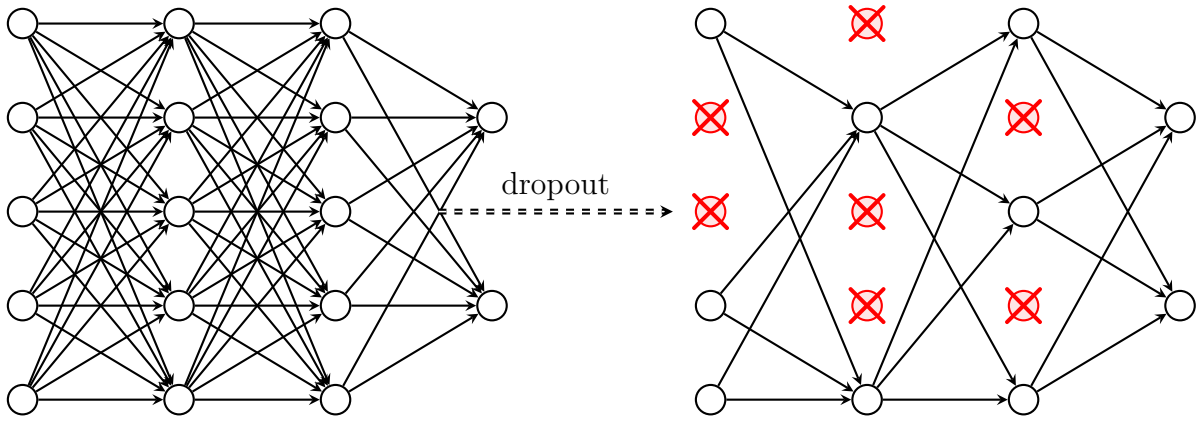


Figure 3.3 – *Representation of how the dropout technique affects a feed-forward neural network architecture. Please remind that this architecture is modified after each mini batch.*

means that the behavior of the network will not change too much if we change randomly some inputs. A regularized network will have difficulties to learn the effect of local noise from the data. The network will learn to predict the data as a whole and will respond only to behavior which is often seen across the training set. Contrariwise, a network with large weights will often change its behavior in response to small changes in the inputs of some individuals from the training set. Therefore, an unregularized network may use large weights to learn information about noise in the training data. In conclusion, regularized networks are constrained to build simpler model based on recurrent patterns in the data set and are robust to noisy data.

In this master thesis we will not use this type of regularization, we will prefer the dropout technique. However, we mention it for completeness.

Dropout regularization

Dropout is a radically different technique for regularization. Unlike L1 and L2 regularization, dropout does not modify the loss function. Instead, the technique modifies the network structure itself. We first focus on how dropout works for a classic neural network and explain the difference for recurrent neural networks after that.

We start by randomly (and temporarily) deleting a proportion $p \in [0, 1)$ of the hidden neurons in the network, while leaving the input and output neurons untouched. We give the network a mini-batch size of learning sample, on this mini-batch we compute the loss function and its gradient. Then, we update the weights (only the non-deleted ones). Afterward, we repeat the same process, we first restore the neurons which were dropped out, then we choose a new random subset of hidden neurons to delete. We estimate the gradient for this mini-batch, and we update the weights of the network. Figure 3.3 presents an example of how a neural network is modified after some neurons are dropped. We repeat

this process again and again, and our network will learn in this way a set of weights. Of course, this set of weights will have been learnt under conditions in which a proportion p of the hidden neurons were dropped out. When we will run the full network more layers will be active. To balance for that, we will scale all the weights outgoing from the hidden neurons by p.

Why will this technique prevent overfitting? If we stop thinking about dropout, but instead we think about training a classic neural network without regularization. Consider that we train several neural networks on the same data set; these networks may not start with the same initial weights and thus give different results. We could consider using an averaging system over all the trained networks to make a prediction. This kind of averaging scheme is often found to be a powerful (though expensive) way of reducing overfitting. The reason is that the different networks may overfit in different directions, and averaging may help to prevent this kind of overfitting.

Then, when we train a network for several epochs using dropout, we are training different neural networks. Thus, the dropout procedure is like averaging the prediction of many different networks. These different networks will overfit in different directions, and the dropout procedure is likely to reduce this overfitting.

In [20], they said about dropout: "*This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons*". In other words, the dropout procedure ensures that the network is robust with respect to the loss of any individual piece of evidence. We see then the similarity to L1 and L2 regularization, which tend to reduce weights, and make the network more robust to the loss of any individual connection in the network.

In RNN we do not apply dropout for all the weights in the recurrent units. Instead, we simply apply this technique to the last weights of the layer, the input weights of the next recurrent layer. Thus, a proportion p of the outputs from the recurrent layer will not be considered in the next layer. The philosophy is the same, the network will learn to adapt itself to different situations and will be more general.

In this master thesis we will prefer dropout over L1 and L2 regularization, it is easily applicable and is sufficient to control overfitting. For RNN in practice, the dropout rate is generally not larger than 0.2. Again, there exist no mathematical rules to choose this rate.

3.3.4 Bidirectional LSTM

Bidirectional recurrent neural networks (BRNN) were first presented in [37]. Bidirectional LSTM (Bi-LSTM) train two parallel LSTM instead of one in the input sequence. The first one is trained on the input sequence as-is while the second is trained on the reverse input sequence. The building of Bi-LSTM is straightforward. Bi-LSTM are more powerful than

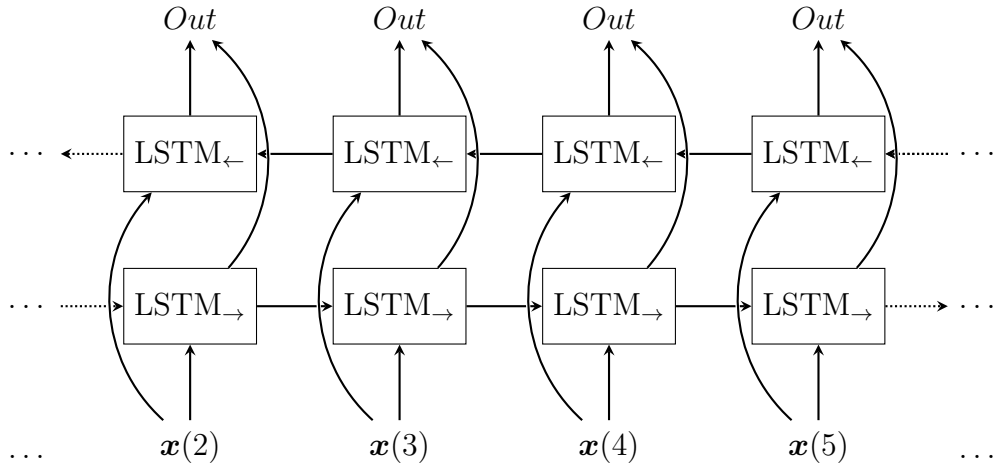


Figure 3.4 – *Representation of a bidirectional LSTM layer.*

classic LSTM since they can be trained using all available information on the past and the future for a particular time-step; hence, the past and future dependency information are both exploited to capture the temporal information. Thus, for each time step of the input sequence the network gives two different outputs and these can be averaged. The training of a Bi-LSTM network is done in the same way as a classic LSTM since the two duplicates LSTM do not have any interactions. For a representation of a Bi-LSTM see Figure 3.4.

3.4 SVDD in R

Since there is no existing package in R to create SVDD, we have implemented it. We use the function *solve.QP* from the package *quadprog* to solve the dual problem 2.19 in R. Our implementation allows us to use any kernel defined as a dot product. For a mathematical description of the different kernels implemented and used in practice, we refer you to the appendix B.

Chapter 4

Anomaly detection on a real world data set

In this chapter, we will apply the statistical techniques presented in the first three chapters to predict extremely rare events on a real data set. Some previous results were obtained on this data set in the literature. Our goal is to compare our LSTM-SVDD anomaly detection with the parametric techniques used in [24] and [25] assuming normality of the errors. Moreover, we will compare our results with the previous results on the same data set obtained in [31]. The data set comes from [31] and can be downloaded freely using the link in the article.

4.1 Data description

The data set comes from a pulp and paper manufacturing industry. It is a multivariate time series process and contains a rare event of paper break. The data contains sensor readings at regular time intervals and a variable to see if the event occurs or not. It is unwanted that the paper breaks so we want to prevent it. However, the techniques that we will use will not allow us to find precisely the variables implying this event.

Several sensors are placed at different places in the machine; they measure both materials and process variables. If the paper breaks, the entire process is stopped until the problem is found. It can take more than one hour to find it. Even if we reduce the break events by only 5%, we will significantly reduce the costs. The objective is to predict the break in advance to be able to identify the possible cause and repair it before total break.

This data set contains 18398 ($= T$) records at different time steps; all these records are coming from a single machine. Each of these records has the following variables:

- time: the timestamp of the record

- y: the binary response variable indicating if the machine is broken ($y = 1$) or not ($y = 0$). There are only 124 records with $y = 1$.
- x_1-x_{64}: the predictor variables. All of these variables are continuous but x_{28} and x_{61}. x_{28} is a categorical variable, while x_{61} is binary. The variable x_{28} is related to the type of paper used in the machine. We have no other available description of these variables for data anonymity.

4.2 Challenges and existing results

The break event is extremely difficult to predict several time steps ahead because this event is generally instant. However, being able to predict a failure is very important to prevent it proactively. Then, we should shift the event occurrence variable for one or two time steps to formulate an early classification problem. Moreover, we do not have access to a huge amount of break events since the data set is highly imbalanced. These are the two main challenges we will need to overpass to build a model predicting break events.

Since we consider a highly imbalanced data set we, will consider the F_1-score as performance measure. The F_1-score is the harmonic mean of precision and recall. For this problem, precision is defined as the fraction of break events among the predicted break events, while recall is the fraction of break events correctly predicted as such.

Definition 4.1. Consider a binary classification where the targets are $\{y_1, \dots, y_n\}$, and the predicted values are $\{\hat{y}_1, \dots, \hat{y}_n\}$. We define the recall, the precision and the F_1-score as

$$\begin{aligned} precision &= \frac{\#(\hat{y}_i = 1 | y_i = 1)}{\#(\hat{y}_i = 1)}, \\ recall &= \frac{\#(\hat{y}_i = 1 | y_i = 1)}{\#(y_i = 1)}, \\ F_1 &= 2\frac{precision \cdot recall}{precision + recall}. \end{aligned}$$

In [31], they used XGBoost and AdaBoost models; combining these techniques with feature engineering they reach a F1-score of 0.082 for a prediction one time step ahead, and a F_1-score of 0.114 to predict up to two time steps ahead.

4.3 Explanatory data analysis

In this section, we will explore several variables from the data, including continuous and categorical variables. The correlation between variables will also be inspected.

Figure 4.1 represents the distribution of the paper break event variable. We can see that the data set is indeed highly imbalanced. Figure 4.2 shows the distribution of the two categorical variables. We see that the binary variable x_{61} is also an extreme rare event. From the distribution of the paper type variable x_{28} we conclude that some types of paper are used more often than others. It is plausible that a less usual type of paper has more chance to provoke a break event.

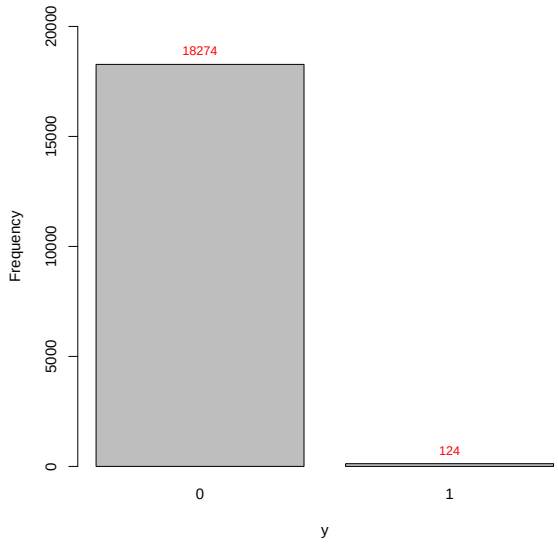


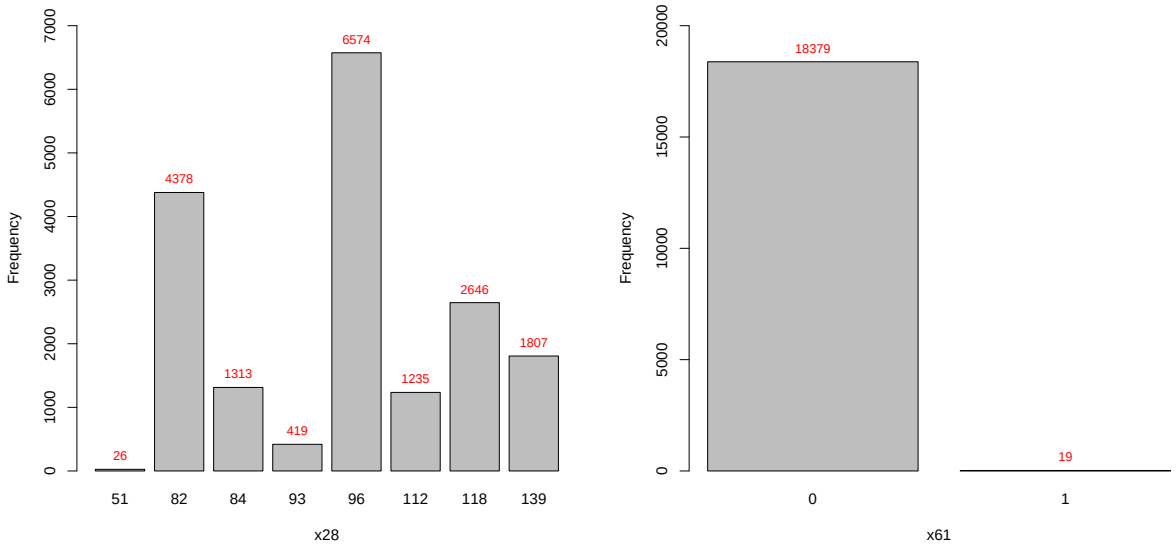
Figure 4.1 – *Break event variable (y, response variable).*

Figure 4.3 represents the correlation between the continuous predictor variables with a significance level of 0.5. It means that only the pairs of variables with an absolute correlation of at most 0.5 are displayed. A blank case does not mean that the correlation is equal to 0, but that the correlation does not met the significance level. To analyze the highest correlations, Figure 4.4 illustrates the ones with a significance level of 0.75.

We see from these two figures that some variables of our database are highly correlated. Even if collinearity is not a problem for neural networks, in contrast to linear regression, it will allow us to reduce the number of predictor variables.

Figure 4.5 represents the predictor variable x_{60} as a function of the time step, while Figure represents 4.6 the variable x_{59}. Finally, Figure 4.7 shows the predictor variable x_6 as a function of the time step.

Since there are many variables, we will not display the graphics of all predictor variables, but we want to give some hints on the distribution of some variables. From these three figures we see that these variables have different patterns. x_{60} has a descending pattern over time, which makes this variable probably non-stationary, we will discuss more in detail about stationarity in the next section. In contrary, the variable x_{59} seems constant over time excluding some abrupt peaks which may be related to some break events. Finally,



(a) *Paper type variable* x_{28}. (b) *Binary variable* x_{61}.

Figure 4.2 – *Distribution of the categorical predictor variables.*

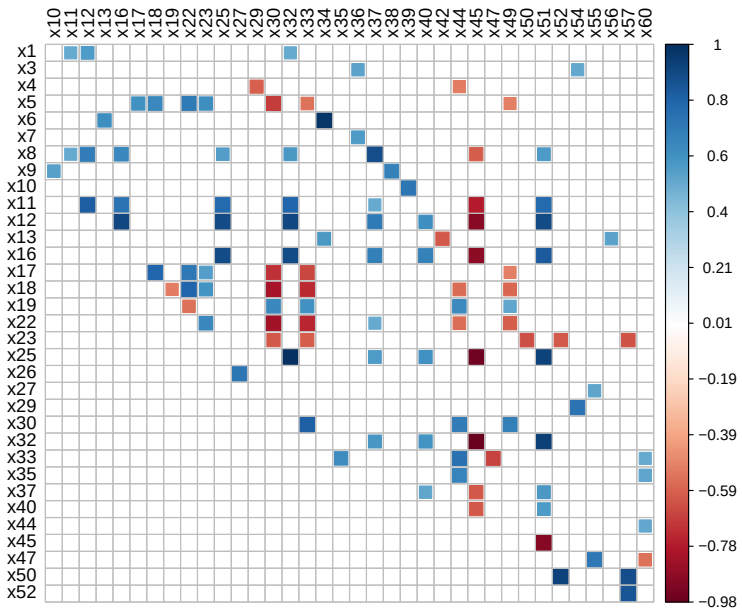


Figure 4.3 – *Correlation between the continuous predictor variables with a significance level of 0.5.*

the variable x_6 is not constant but does not show many values far from its mean over time neither. In the next section, we will see that almost all variables are non-stationary and why we should make these variables stationary before feeding them to the LSTM network.

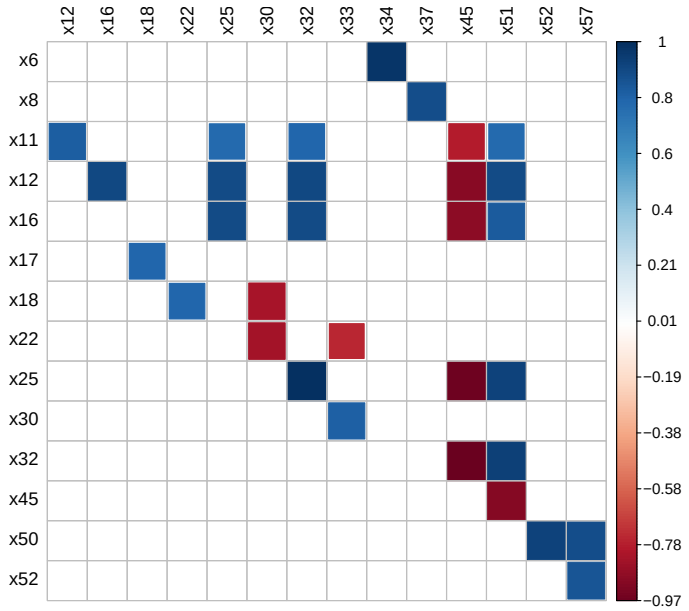


Figure 4.4 – *Correlation between the continuous predictor variables with a significance level of 0.75.*

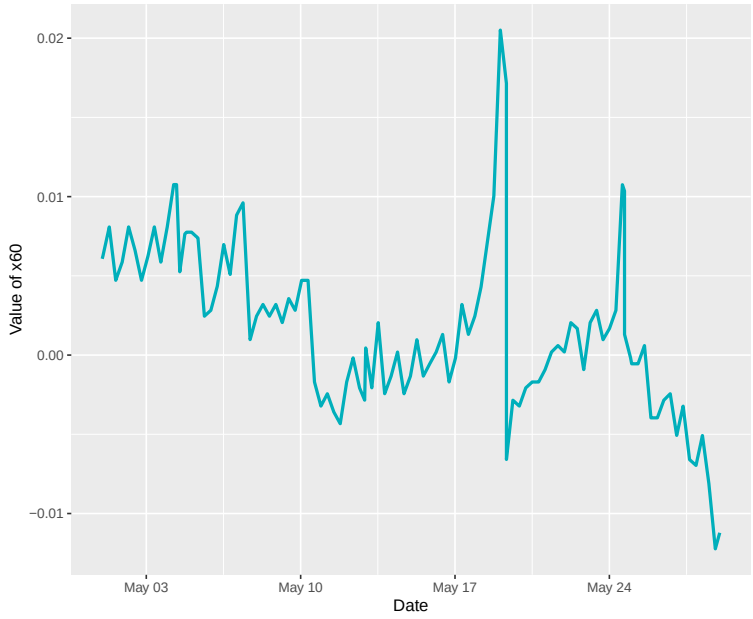


Figure 4.5 – *Predictor variable x_{60} as a function of the time step.*

4.4 Data pre-processing

In this section, we will first discuss about the stationarity of the continuous variables. Afterward, we will apply a dimension reduction technique, the principal component analysis.

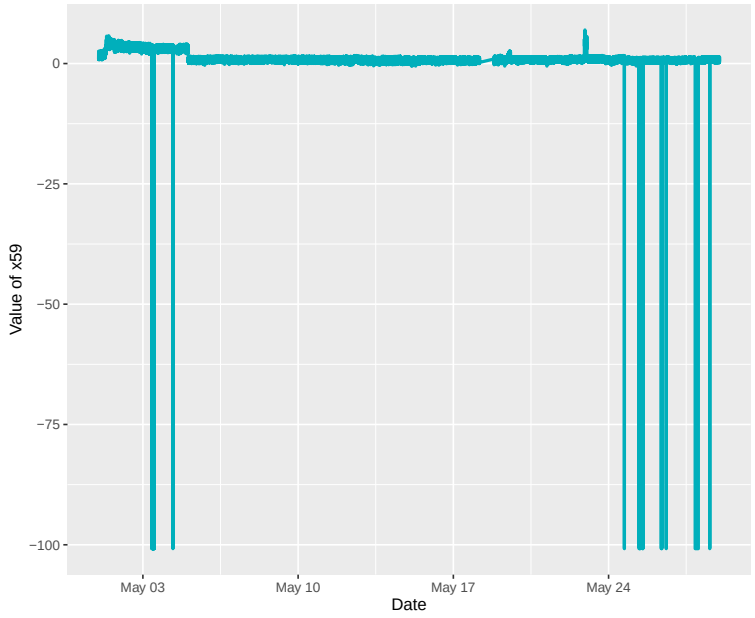


Figure 4.6 – *Predictor variable x_{59} as a function of the time step.*

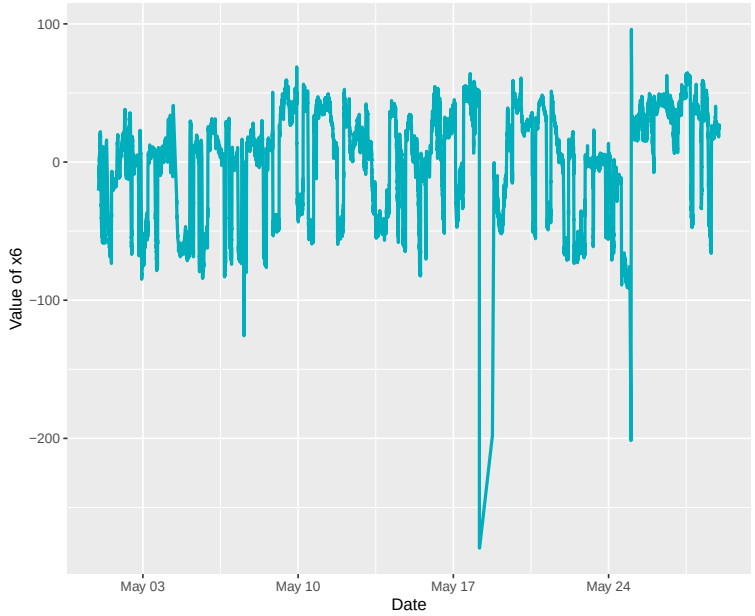


Figure 4.7 – *Predictor variable x_6 as a function of the time step.*

Finally, we will transform the problem to an early detection problem.

To verify our intuition obtained by observing the graphs in the previous section, we will test the stationarity of the time series. We use the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test, which tests the null hypothesis that the time series is stationary, for a

random walk alternative (see [15]). More precisely, the test considers three different null hypotheses, the trend stationarity, the level stationarity and the zero mean stationarity. If the time series is trend stationary, the values of the time series are time-dependent. Mathematically, a trend stationary time series $\boldsymbol{x}$ is written as

$$x(t) = a + bt + \varepsilon(t),\ t = 1, \ldots, T,$$

where $\varepsilon(t)$ follows a zero-mean distribution. Then, we need the time series to be level stationary ($b = 0$ in the above equation). If it is not the case, our LSTM model will learn patterns in the data at a certain time which will no longer be accurate in the future. It implies that the reconstruction error on future points (with a different pattern) will be large and every point will be detected as an anomaly even if it is normal. The anomaly detection will become useless.

Performing the KPSS test on each of the continuous variables, we find that the null hypothesis of level stationarity is rejected for 38 variables (at a significance level of 0.05), while for 21 continuous variables it was not rejected. For the majority of the considered continuous variables the level stationarity is not achieved, in this situation the LSTM model will learn many time-dependent patterns. We tried to make the data stationary by differencing all variables from their previous time step. Moreover, when a break occurs, the machine is stopped until it is repaired. It implies that each time step following a break event is the beginning of a new run from the machine; thus, the chance is great that the difference between this time step and the previous one will be huge (since they have not run under the same regime). However, we do not want that our model predicts this difference as anomalous (since the break has already occurred); so, we set this difference to 0. Mathematically, we transform our data set as

$$x_i^{(1)}(1) = 0,$$

and

$$x_i^{(1)}(t) = \begin{cases} x_i(t) - x_i(t-1) & \text{if } y(t-1) = 0, \\ 0 & \text{otherwise,} \end{cases}$$

$t = 2, \ldots, T$, $i = 1, \ldots, 61$. From this point we consider the data set $\{\boldsymbol{x}_1^{(1)}, \ldots, \boldsymbol{x}_{61}^{(1)}, \boldsymbol{y}\}$.

As in [25] or [41], we use a principal component analysis to reduce the number of continuous variables. This decision is motivated by two main reasons: the first is that we have seen some variables highly correlated to another. Thus, we can reduce the number of variables without losing much information. The second one is that the less variables are considered, the faster the support vector data description will be optimized.

We found that 35 principal components are enough to explain a cumulative proportion of 90.9% of the variance . In addition to these 35 principal components, we consider the two categorical variables $\boldsymbol{x}_{28}^{(1)}$ and $\boldsymbol{x}_{61}^{(1)}$; our data set is now composed of 35 continuous, and 2 categorical predictor variables associated to a response binary variable $\boldsymbol{y}$ for each of the T time steps.

For each univariate predictor time series from this new data set, we perform the KPSS test; we conclude that the null hypothesis cannot be rejected (at a significance level of 0.05, individually for each time series). Even if these results do not prove that our time series are level stationary, we do not have enough information to reject this hypothesis. Then, a LSTM will probably not learn too many time-dependent patterns on this data.

The last step is to transform our data set to perform an early classification problem. As suggested in [31], we tried to predict the breaks up to two time steps ahead (4 minutes). To do so, we detect the break time steps in the data and consider as a break the two previous time steps (by setting the variable y to 1). Finally, since we are not interested in predicting the break once it has occurred, we delete the sample coming from an instantaneous break event from our data set (the original time steps with $y = 1$). Our data base is now composed of 37 predictor variables for 18,274 time steps. There are now 248 records with $y = 1$; our goal is to detect the 124 break events either one or two time steps ahead.

This final data base will be used in the following sections for anomaly detection.

4.5 LSTM-based anomaly detection

In this section, we will consider the anomaly detection as described in section 3.1. Different values for l and m will be used, thus we consider l time steps in the past to predict m time steps in the future.

First, we consider a Bi-LSTM model trained with $l = 5$ and $m = 1$. Afterward, we will consider two other Bi-LSTM models with $l = 10$ and $m = 3$ or 5.

For each optimal Bi-LSTM network, we will compare the performance between the anomaly detection with SVDD and the anomaly detection as in [25], where the errors are assumed to follow a multivariate Gaussian distribution.

We divide our complete set of sub-time series into 4 subsets, a training set (50% of the complete data-set), a first validation set (10%), a second validation set (20%), and finally a test set (20%). Since we deal with time series, the set is not divided at random but by keeping the temporal order. Moreover, each of these subsets can be divided into a normal (containing the normal sub-time series) and an anomalous subset.

For each considered combination of parameters (l, m), we tune the Bi-LSTM network by k-fold cross validation (adapted to time series) on the normal training set. We prefer Bi-LSTM over classic LSTM since we want to use both past and future information. Figure 4.8 displays how the tuning of the Bi-LSTM parameters is done using cross-validation, and the model giving the lowest error is chosen.

When we assume normality of the error distribution, we use the first validation set to estimate the parameters of the normal distribution and we choose the threshold τ as the one maximizing the F_1-score on the second validation set. Finally, we assess the performance of our model by training the LSTM network on the training and first validation sets, using the

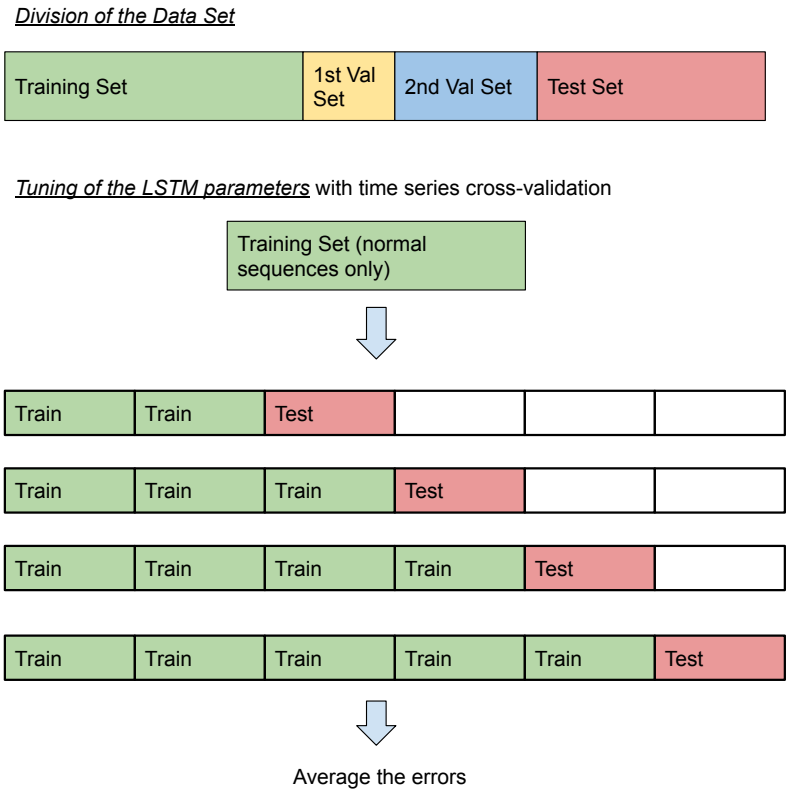


Figure 4.8 – *How the parameters for the Bi-LSTM are tuned using cross-validation adapted to time series.*

second validation set to estimate the parameters of the normal distribution, and assessing the performance on the test set with the best threshold τ. This procedure is summarized on Figure 4.9.

When we use the SVDD anomaly detection, the procedure is trickier. We use the first validation set with cross-validation to choose the best parameters for each considered kernel. The used kernels are described in the appendix, and we need to tune the parameters of the kernel and the cost parameter C for the SVDD to reach the best performance. When we train a SVDD on a set of normal errors, we only use approximately 400 error vectors for time computing reasons. Moreover, in [40] and [21] only 100 samples are used for the training set for an 256-dimensional problem, and we can then describe high-dimensional data with few individuals.

When we have found the best parameters for each kernel, we assess the performance of our model by training a SVDD with 400 normal error vectors from the second validation set and we test it on the test set. This procedure is described on the figures 4.10 and 4.11.

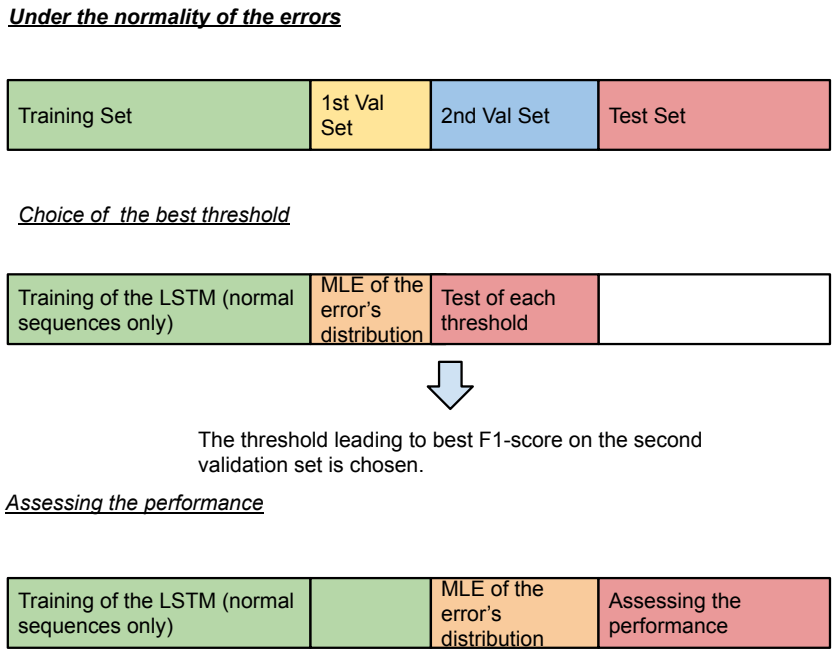


Figure 4.9 – *Construction and test of the model under the assumption of normality for the error vectors.*

4.5.1 Bi-LSTM with l=5 and m=1

We find that the optimal topology is composed of two bidirectional hidden layers. The first hidden layer has 35 units, while the second has 30 units. Each of these layers have an associated dropout rate of 0.05. We used the *Adam* optimizer to train this network with a batch size equal to 4.

For each kernel, we fix a grid of parameters to test from which we will extract the best ones. The grid for the parameter values are the following:

- For the Gaussian kernel, we test the values $\sigma \in \{0.5 + i \mid i = 0, 1, \ldots, 49\}$.
- For the Laplacian kernel, we test the values $\sigma \in \{2^{i/2} \mid i = 0, 1, \ldots, 40\}$.
- For the circular kernel, we test the values $\sigma \in \{i/2 \mid i = 10, 11, \ldots, 50\}$.
- For the Cauchy kernel, we test the values $\sigma \in \{2^{i/2} \mid i = 0, 1, \ldots, 24\}$.
- For the Generalized Histogram Intersection kernel, we test the values $(\alpha, \beta) \in \{(i/4, j/4) \mid i, j = 1, 3, 5, \ldots, 21\}$.

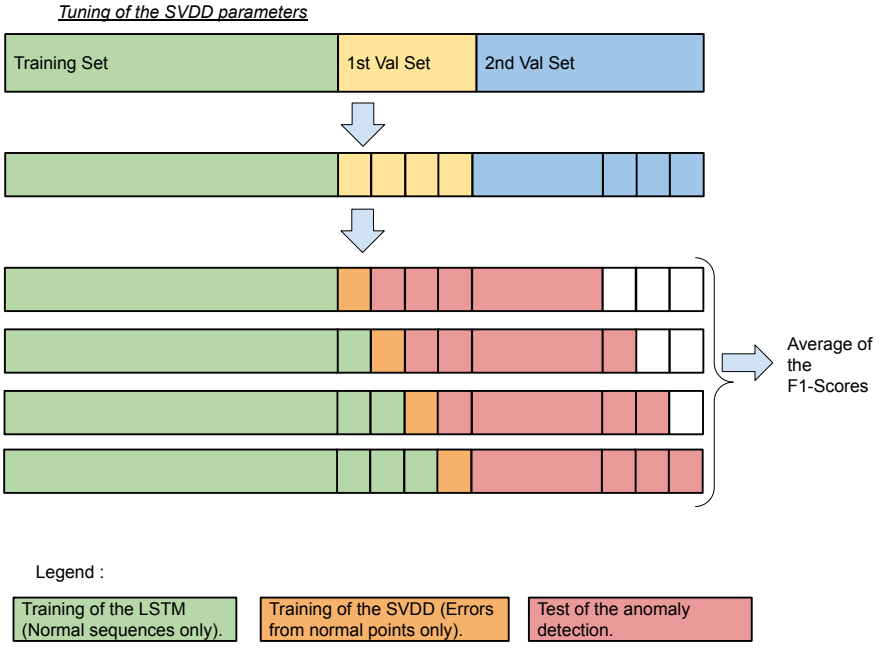


Figure 4.10 – *How the parameters for the SVDD are tuned using cross-validation adapted to time series. We choose the parameters leading to the greatest F_1-score.*

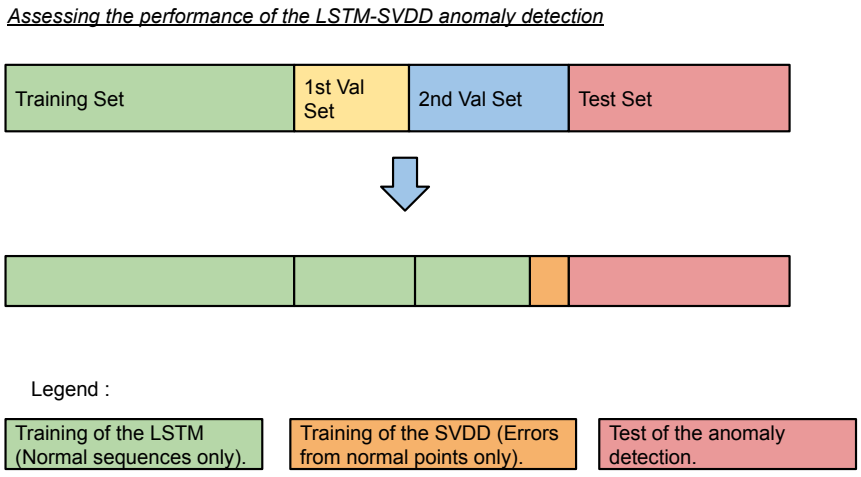


Figure 4.11 – *How we assess the performance of our LSTM-SVDD anomaly detection model.*

Once the best parameter for the corresponding kernel is defined, we tune the value of the cost parameter C for the SVDD. The best value is determined by testing the values $C \in \{2^i \mid i = -7, -6, \ldots, 1\}$.

Table 4.1 resumes the optimal parameters found for each kernel.

Kernel	Gaussian	Laplacian	Circular	Cauchy	Histogram
Parameters (σ or (α, β))	13.5	16384	24.5	90.51	$(4.75, 4.25)$
Cost Parameter (C)	0.250	0.125	0.008	0.500	0.250

Table 4.1 – *Optimal parameters for the different kernels leading to the best average F_1-score on the second validation set.*

The optimal value for the threshold under normality assumption is given by $\tau = 191.39$.

Using these optimal parameters, we assess the performance of our model on the test set; the results are the average of the performances obtained by running 10 independent tests on the same set. We precise the standard deviation between brackets. Table 4.2 presents the results obtained for each kernel when we use SVVD, while Table 4.3 presents the results under normality assumption.

Kernel	Gaussian	Laplacian	Circular	Cauchy	Histogram
Recall	0.147 (0)	0.176 (0)	0.029 (0)	0.059 (0)	0.088 (0)
Precision	0.049 (0.004)	0.047 (0.003)	0.020 (0)	0.042 (0.002)	0.049 (0.006)
F_1-score	0.074 (0.004)	0.074 (0.004)	0.024 (0)	0.049 (0.001)	0.063 (0.005)

Table 4.2 – *Values of the mean (standard deviation) of the different statistics for each kernel. These values were found by running 10 independent tests on the same set.*

	Recall	Precision	F_1-score
Under normality	0.088 (0)	0.052 (0.004)	0.065 (0.003)

Table 4.3 – *Values of the mean (standard deviation) of the different statistics under normality assumption. These values were found by running 10 independent tests on the same set.*

From these results, we see that the Gaussian and Laplacian kernels produce the best performances. Moreover, the Laplacian kernel has a higher recall but a lower precision than the Gaussian kernel. The circular and Cauchy kernels do not perform well for this problem. We see that the the performance under normality assumption is worse than when we use a good kernel associated with a SVDD data description.

4.5.2 Bi-LSTM with l=10 and m=3

When we face a many-to-many sequences prediction problem with a LSTM architecture, we need to use an Encoder-Decoder structure to adapt the number of time steps from the network output to the target's number of time steps. The mathematical details are given in the appendix C.

We find that the optimal topology is composed of two bidirectional hidden layers. The first hidden layer has 35 units, while the second has 30 units. Each of these layers have an associated dropout rate of 0.07. We used the *Adam* optimizer to train this network with a batch size equal to 4.

Table 4.4 summarizes the optimal parameters found for each kernel.

Kernel	Gaussian	Laplacian	Circular	Cauchy	Histogram
Parameters (σ or (α, β))	31.5	32768	17.5	181.02	$(5.25, 4.25)$
Cost Parameter (C)	0.500	0.500	0.008	0.250	0.250

Table 4.4 – *Optimal parameters for the different kernels leading to the best average F_1-score on the second validation set.*

The optimal value for the threshold under normality assumption is given by $\tau = 1601.25$.

Using these optimal parameters the same procedure as in the previous section is used and the tables 4.5 and 4.6 summarize the results.

Kernel	Gaussian	Laplacian	Circular	Cauchy	Histogram
Recall	0.118 (0)	0.118 (0)	0.176 (0)	0.118 (0)	0.059 (0)
Precision	0.057 (0.001)	0.062 (0.001)	0.068 (0.001)	0.057 (0.001)	0.039 (0.003)
F_1-score	0.078 (0.001)	0.081 (0.001)	0.098 (0.002)	0.077 (0.001)	0.047 (0.002)

Table 4.5 – *Values of the mean (standard deviation) of the different statistics for each kernel. These values were found by running 10 independent tests on the same set.*

In this case, we see that the performance has increased for the Gaussian and the Laplacian kernel with respect to the previous results. Moreover, for these two kernels we see that the stability of the results is better than in the previous case. The Histogram Generalized Intersection kernel still performs poorly. Both the Cauchy and circular kernels have their performances increased a lot when we consider more prediction in the future. Furthermore, the circular one seems to be the most adapted for this situation. Finally, we see that for the

	Recall	Precision	F_1-score
Under normality	0.088 (0)	0.047 (0)	0.061 (0)

Table 4.6 – *Values of the mean (standard deviation) of the different statistics under normality assumption. These values were found by running 10 independent tests on the same set.*

Gaussian, Laplacian and Cauchy kernels the recall is not better than the previous score, but their global F_1-score is better due to their better precision. Then, increasing the values for l and m is a way to increase the precision of the prediction.

Finally, under the assumption of normality for the errors we do not see an improvement in the performances and the data description using kernels and SVDD still performs better than this parametric assumption.

4.5.3 Bi-LSTM with l=10 and m=5

From this point, we decide to not consider the Generalized Histogram Intersection kernel anymore due to its poor performance and its prohibitive computational cost.

We find that the optimal topology is composed of two bidirectional hidden layers. Both hidden layers have 35 units. Each of these layers have an associated dropout rate of 0.14. We used the *Adam* optimizer to train this network with a batch size equal to 2.

Table 4.7 summarizes the optimal parameters found for each kernel.

Kernel	Gaussian	Laplacian	Circular	Cauchy
Parameters (σ)	38.5	65536	21.5	181.02
Cost Parameter (C)	0.500	0.500	0.031	1

Table 4.7 – *Optimal parameters for the different kernels leading to the best average F_1-score on the second validation set.*

The optimal value for the threshold under normality assumption is given by $\tau = 3979.80$.

Using these optimal parameters the same procedure as in the two previous sections is used and the tables 4.8 and 4.9 summarize the results.

We see from these results that the performance of all kernels but the Laplacian are worse than the results considering only 3 time steps to predict. However, we see that the recall is very high when we consider the circular kernel, but associated with a high standard deviation. The circular kernel seems not to be a stable kernel in this situation. Moreover,

Kernel	Gaussian	Laplacian	Circular	Cauchy
Recall	0.118 (0)	0.118 (0)	0.347 (0.173)	0.059 (0)
Precision	0.051 (0.002)	0.063 (0)	0.046 (0.008)	0.043 (0)
F_1-score	0.071 (0.002)	0.082 (0)	0.078 (0.011)	0.05 (0)

Table 4.8 – *Values of the mean (standard deviation) of the different statistics for each kernel. These values were found by running 10 independent tests on the same set.*

	Recall	Precision	F_1-score
Under normality	0.088 (0)	0.043 (0)	0.058 (0)

Table 4.9 – *Values of the mean (standard deviation) of the different statistics under normality assumption. These values were found by running 10 independent tests on the same set.*

except for the Laplacian kernel the standard deviation of all the results is greater than the previous ones. We can explain it by the fact that we are trying to predict more time steps in the future by using the same number of time steps as input. Therefore, the LSTM network has more difficulties to learn a pattern in the normal time series and produces this instability in the results. By considering more time steps as input we could certainly further improve the performance and the stability of the results.

Again, we see that the performance is worse when assuming normality of the errors than when using SVDD. Moreover, the more we consider to predict time steps in the future, the worse the performance of this assumption is. We can explain it by the fact that the more we predict time steps, the higher the dimension of the error vectors is. Therefore, the normality assumption is even more unlikely.

4.6 LSTM-based Auto-Encoder-Decoder anomaly detection

In this section, we will consider anomaly detection as described in section 3.2. Different values for the length of the sequence (L) will be considered. The same procedure as described in section 4.5 is used to tune the LSTM, to find the best SVDD parameters or the optimal threshold and to assess the performance.

Table 4.10 summarizes the optimal topology of the LSTM network for each value of L. Please note that we use a simple LSTM and not a Bi-LSTM for this problem.

Table 4.11 summarizes the optimal parameters found for each value of L and for each

	Size of hidden layers	Dropout rate	Optimizer	Batch size
L=2	(35,35)	0.01	*Adam*	4
L=3	(35,35)	0	*Nadam*	4
L=5	(35,30)	0	*Nadam*	8
L=10	(35,30)	0.03	*Adamax*	1

Table 4.10 – *Optimal topology for the different lengths of sequences. The notation* (d_1, d_2) *indicates that the first hidden layer has* d_1 *units and the second* d_2 *units.*

kernel.

	Kernel	Gaussian	Laplacian	Circular	Cauchy
L=2	Parameters (σ)	26.5	2048	13	45.25
	Cost Parameter (C)	0.250	0.500	0.031	0.250
L=3	Parameters (σ)	38.5	65536	21.5	181.02
	Cost Parameter (C)	0.125	0.016	0.008	0.125
L=5	Parameters (σ)	37.5	256	25	64
	Cost Parameter (C)	0.125	0.016	0.008	0.125
L=10	Parameters (σ)	42.5	8192	17	128
	Cost Parameter (C)	0.125	0.500	0.008	0.500

Table 4.11 – *Optimal parameters for each considered length of time sequences and for the different kernels leading to the best average* F_1*-score on the second validation set.*

The optimal values for the threshold under normality assumption for the errors are given in Table 4.12.

Under normality	L=2	L=3	L=5	L=10
Optimal τ	332.86	111.45	694.92	19455.40

Table 4.12 – *Optimal threshold for the different length of considered time series under the assumption of normality of the errors.*

Using these optimal parameters, we run 10 independent tests on the same sets for each considered length of sequences. The results are displayed in the Tables 4.13 and 4.14.

	Kernel	Gaussian	Laplacian	Circular	Cauchy
L=2	Recall	0.268 (0.035)	0.274 (0.034)	0.200 (0.030)	0.271 (0.036)
	Precision	0.076 (0.007)	0.076 (0.007)	0.087 (0.022)	0.076 (0.007)
	F_1-score	0.118 (0.011)	0.118 (0.012)	0.118 (0.026)	0.119 (0.012)
L=3	Recall	0.324 (0.031)	0.3 (0.033)	0.173 (0.017)	0.329 (0.036)
	Precision	0.077 (0.008)	0.084 (0.007)	0.089 (0.017)	0.078 (0.009)
	F_1-score	0.125 (0.013)	0.131 (0.011)	0.117 (0.017)	0.126 (0.014)
L=5	Recall	0.25 (0.056)	0.321 (0.038)	0.176 (0.057)	0.25 (0.056)
	Precision	0.061 (0.008)	0.064 (0.007)	0.064 (0.007)	0.060 (0.008)
	F_1-score	0.097 (0.014)	0.107 (0.010)	0.099 (0.023)	0.097 (0.014)
L=10	Recall	0.226 (0.024)	0.194 (0.025)	0.779 (0.029)	0.826 (0.366)
	Precision	0.059 (0.008)	0.058 (0.008)	0.041 (0.002)	0.027 (0.017)
	F_1-score	0.093 (0.011)	0.089 (0.011)	0.078 (0.003)	0.046 (0.019)

Table 4.13 – *Values of the mean (standard deviation) of the different statistics for each value of L and kernel. These values were found by running 10 independent tests on the same set.*

Under normality	Recall	Precision	F_1-score
L=2	0.132 (0.029)	0.056 (0.0106)	0.079 (0.015)
L=3	0.759 (0.019)	0.0346 (0.001)	0.066 (0.002)
L=5	0.571 (0.032)	0.045 (0.006)	0.083 (0.010)
L=10	0.029 (0)	0.022 (0.001)	0.025 (0.001)

Table 4.14 – *Values of the mean (standard deviation) of the different statistics under normality of the errors assumption for each value of L. These values were found by running 10 independent tests on the same set.*

We can extract some interesting conclusions from these results. First of all, we clearly see that for any considered length of sequences, SVDD with any kernel leads to a better F_1-score than when we consider the assumption of normality for the error vectors. These results prove that the assumption of normality is not realistic for this data set and that using a non-parametric technique like SVDD is required. When we use SVDD the best results are achieved when we consider a sequences length of 3. Considering a length of 3

for the sequences and assuming normality of the error vectors we achieve a good recall. However, the precision is bad and thus leads to a poor F_1-Score. We see then that the non-parametric technique using SVDD leads to a lower recall but a much higher precision and therefore to a better F_1-Score. When we consider a prediction length of 2, the four kernels considered lead approximately to the same F_1-Score. Although, if we consider a prediction length greater than 3, the precision is reduced and thus the F_1-Score. Moreover, when we consider sequences of length 10 we see that both circular and Cauchy kernels have a high recall but a very poor precision. The results for these two kernels are similar to the results under the assumption of normality when we consider a prediction length of 3. We conclude that in a general case, we should consider the Laplacian or the Gaussian kernel which lead to satisfactory and stable results for any considered length, and then are more prone to be generalized to other anomaly detection problems

Finally, we see that our best F_1-score is better than the results given in [31]. It shows that considering an anomaly detection combining both LSTM and SVDD can solve difficult problems and give satisfactory results.

Conclusion

In this thesis, we successfully combined LSTM and SVDD for anomaly detection. The two first chapters mathematically and rigorously present these two techniques. The third chapter sets the basis on anomaly detection. Finally, the fourth and last chapter applies LSTM and SVDD on a real world data set.

The real world data set that we have considered contains a rare break event of a machine coming from a pulp and paper manufacturing industry. Our goal was to predict this event before it happens. We have encountered many difficulties while working on this data set. First, we have seen that the data was not stationary. This non-stationarity implies that a LSTM neural network will learn time dependent patterns and makes the anomaly detection useless. Thus, we differentiate the data with respect to the time steps. Afterward, since some variables from our data set were highly correlated, we applied a principal components analysis to reduce the dimensionality of the problem. Finally, we transformed the problem into an early anomaly detection problem by shifting the break events by one and two time steps.

We have shown that using a non-parametric data description, the SVDD allowed us to reach better results than considering a normality assumption for the error vectors on our real world data set. Therefore, when a LSTM-based anomaly detection is used we have no guarantee that the errors follow or not a Gaussian distribution and we should consider the use of SVDD. Considering SVDD the best F_1-score reached is 0.131, while the best F_1-score is only 0.083 assuming normality of the errors. Moreover, our LSTM-SVDD anomaly detection technique has increased the F_1-score reached by the authors of [31] using XGBoost and AdaBoost models (0.114). The best performance has been achieved by using an Auto-Encoder-Decoder LSTM.

There is much that remains to be explored about the technique presented in this thesis. We should test the LSTM-SVDD anomaly detection on other real world data sets to see how this technique becomes generalized on different sets. Moreover, considering the computer on which the simulations were done, the training of the SVDD was limited to few individuals. In future works we should consider training the SVDD on more individuals to see if it increases the performance.

Appendices

Appendix A

Optimizers: Adam, Adamax and Nadam

This appendix is based on [19] and [33]. Only the algorithms are given, for mathematical details on how there are constructed we refer the reader to the mentioned papers.

In this section, we denote the set of weights a at certain step of the training t, by $\boldsymbol{W}(t)$. The error function with respect to the weights at step t is denoted $E(\boldsymbol{W}(t))$, and its gradient with respect to this set of weights $\nabla_{\boldsymbol{W}(t)} E(\boldsymbol{W}(t))$.

Algorithm 1: *Adam* algorithm. Recommended initial parameters are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$. $\boldsymbol{g}(t)^2$ denotes the element-wise product $\boldsymbol{g}(t) \otimes \boldsymbol{g}(t)$. All operations on vectors are element-wise.

Input: α the learning rate, $\beta_1, \beta_2 \in [0, 1)$ the exponential rates decay, ε, $E(\boldsymbol{W})$ the error function, and $\boldsymbol{W}(0)$ the initial weight vector.

Output: $\boldsymbol{W}(T)$ the final weight vector.

$\boldsymbol{m}(0) \leftarrow 0$;

$\boldsymbol{v}(0) \leftarrow 0$;

$t \leftarrow 0$;

while *the stopping criterion is not met* **do**

$t \leftarrow t + 1$;

$\boldsymbol{g}(t) \leftarrow \nabla_{\boldsymbol{W}(t-1)} E(\boldsymbol{W}(t-1))$;

$\boldsymbol{m}(t) \leftarrow \beta_1 \cdot \boldsymbol{m}(t-1) + (1 - \beta_1) \cdot \boldsymbol{g}(t)$;

$\boldsymbol{v}(t) \leftarrow \beta_2 \cdot \boldsymbol{v}(t-1) + (1 - \beta_2) \cdot \boldsymbol{g}(t)^2$;

$\hat{\boldsymbol{m}}(t) \leftarrow \boldsymbol{m}(t) / (1 - \beta_1^t)$;

$\hat{\boldsymbol{v}}(t) \leftarrow \boldsymbol{v}(t) / (1 - \beta_2^t)$;

$\boldsymbol{W}(t) \leftarrow \boldsymbol{W}(t-1) - \alpha \cdot \hat{\boldsymbol{m}}(t) / (\sqrt{\hat{\boldsymbol{v}}(t)} + \varepsilon)$;

end

$T \leftarrow t$: **return** $\boldsymbol{W}(T)$.

Algorithm 2: *Adamax* algorithm. Recommended initial parameters are $\alpha = 0.002$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. All operations on vectors are element-wise.

Input: α the learning rate, $\beta_1, \beta_2 \in [0, 1)$ the exponential rates decay, $E(\boldsymbol{W})$ the error function, and $\boldsymbol{W}(0)$ the initial weight vector.

Output: $\boldsymbol{W}(T)$ the final weight vector.

$\boldsymbol{m}(0) \leftarrow 0$;

$\boldsymbol{u}(0) \leftarrow 0$;

$t \leftarrow 0$;

while *the stopping criterion is not met* **do**

$t \leftarrow t + 1$;

$\boldsymbol{g}(t) \leftarrow \nabla_{\boldsymbol{W}(t-1)} E(\boldsymbol{W}(t-1))$;

$\boldsymbol{m}(t) \leftarrow \beta_1 \cdot \boldsymbol{m}(t-1) + (1 - \beta_1) \cdot \boldsymbol{g}(t)$;

$\boldsymbol{u}(t) \leftarrow \max(\beta_2 \cdot \boldsymbol{u}(t-1), |\boldsymbol{g}(t)|)$;

$\boldsymbol{W}(t) \leftarrow \boldsymbol{W}(t-1) - (\alpha / (1 - \beta_1^t)) \cdot \boldsymbol{m}(t) / \boldsymbol{u}(t)$;

end

$T \leftarrow t$;

return $\boldsymbol{W}(T)$.

Algorithm 3: *Nadam* algorithm. Recommended initial parameters are $\alpha = 0.002$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-7}$. $\boldsymbol{g}(t)^2$ denotes the element-wise product $\boldsymbol{g}(t) \otimes \boldsymbol{g}(t)$. All operations on vectors are element-wise.

Input: α the learning rate, $\beta_1, \beta_2 \in [0, 1)$ the exponential rates decay, ε, $E(\boldsymbol{W})$ the error function, and $\boldsymbol{W}(0)$ the initial weight vector.

Output: $\boldsymbol{W}(T)$ the final weight vector.

$\boldsymbol{m}(0) \leftarrow 0$;

$\boldsymbol{v}(0) \leftarrow 0$;

$t \leftarrow 0$;

while *the stopping criterion is not met* **do**

$t \leftarrow t + 1$;

$\boldsymbol{g}(t) \leftarrow \nabla_{\boldsymbol{W}(t-1)} E(\boldsymbol{W}(t-1))$;

$\boldsymbol{m}(t) \leftarrow \beta_1 \cdot \boldsymbol{m}(t-1) + (1 - \beta_1) \cdot \boldsymbol{g}(t)$;

$\boldsymbol{v}(t) \leftarrow \beta_2 \cdot \boldsymbol{v}(t-1) + (1 - \beta_2) \cdot \boldsymbol{g}(t)^2$;

$\hat{\boldsymbol{m}}(t) \leftarrow \boldsymbol{m}(t) / (1 - \beta_1^t)$;

$\hat{\boldsymbol{v}}(t) \leftarrow \boldsymbol{v}(t) / (1 - \beta_2^t)$;

$$\boldsymbol{W}(t) \leftarrow \boldsymbol{W}(t-1) - \frac{\alpha}{\sqrt{\hat{\boldsymbol{v}}(t)} + \varepsilon} \left(\beta_1 \cdot \hat{\boldsymbol{m}}(t) + \frac{1 - \beta_1}{1 - \beta_1^t} \cdot \boldsymbol{g}(t) \right) ;$$

end

$T \leftarrow t$;

return $\boldsymbol{W}(T)$.

Appendix B

Kernel functions

Definition B.1. The Gaussian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{y}||^2}{2\sigma^2}\right),$$

where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$, and $\sigma \in \mathbb{R}$.

Definition B.2. The Laplacian kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{y}||}{\sigma}\right),$$

where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$, and $\sigma \in \mathbb{R}^+$.

Definition B.3. The Circular kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} \frac{2}{\pi}\arccos(-\frac{||\boldsymbol{x} - \boldsymbol{y}||}{\sigma}) - \frac{2||\boldsymbol{x} - \boldsymbol{y}||}{\pi\sigma}\sqrt{1 - \left(\frac{||\boldsymbol{x} - \boldsymbol{y}||}{\sigma}\right)^2} & \text{if} ||\boldsymbol{x} - \boldsymbol{y}|| < \sigma, \\ 0 & \text{otherwise,} \end{cases}$$

where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$, and $\sigma \in \mathbb{R}^+$.

Definition B.4. The Cauchy kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{1 + \frac{||\boldsymbol{x} - \boldsymbol{y}||^2}{\sigma^2}},$$

where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$, and $\sigma \in \mathbb{R}$.

Definition B.5. The Generalized Histogram Intersection kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{p} \min(|x_i|^\alpha, |y_i|^\beta),$$

where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$, and $\alpha, \beta \in \mathbb{R}$.

Appendix C

LSTM Encoder-Decoder forward pass

Consider a problem where the input time series are in the form of $\{\boldsymbol{x}(1), \ldots, \boldsymbol{x}(T_1)\} \in \mathbb{R}^{p_1}$, and the target time series are in the form of $\{\boldsymbol{y}(1), \ldots, \boldsymbol{y}(T_2)\} \in \mathbb{R}^{p_1}$, $T_1, T_2 > 0$. Consider an Encoder-Decoder with 2 hidden LSTM layers, let d_1 (resp. d_2) be the number of LSTM memory blocks of the first hidden layer (resp. second hidden layer). Consider that each memory block has a single memory cell, and each memory block is connected to all memory blocks (including itself) in the same hidden layer.

For any $m \in \{c, in, out, \varphi\}$, and $t \in \{1, \ldots, T\}$, the net inputs at time t of the first layer are computed with:

$$\boldsymbol{net}_m^{(1)}(t) = \boldsymbol{W}_m^{(1)}\boldsymbol{x}(t) + \boldsymbol{U}_m^{(1)}\boldsymbol{a}_c^{(1)}(t-1),$$

where, $\boldsymbol{W}_m^{(1)} \in \mathbb{R}^{d_1 \times p_1}$ are the input weight matrices, and $\boldsymbol{U}_m^{(1)} \in \mathbb{R}^{d_1 \times d_1}$ are the recurrent weight matrices.

The activation is computed with:

$$a_{m_j}^{(1)}(t) = f_m^{(1)}(net_{m_j}^{(1)}(t)),$$

$j \in \{1, \ldots, d_1\}$, $m \in \{in, out, \varphi\}$, $t \in \{1, \ldots, T_1\}$. Where, $f_{in}^{(1)}$, $f_{out}^{(1)}$, $f_{\varphi}^{(1)}$ are respectively the activation functions of the input, output and forget gates.

We compute the internal state of each block as:

$$S_{c_j}^{(1)}(0) = 0;\ S_{c_j}^{(1)}(t) = a_{\varphi_j}^{(1)}(t)S_{c_j}^{(1)}(t-1) + a_{in_j}^{(1)}(t)g^{(1)}(net_{c_j}^{(1)}(t)),$$

$j \in \{1, \ldots, d_1\}$, $t \in \{1, \ldots, T_1\}$. Where, $g^{(1)}$ is a differentiable function.

The memory blocks outputs are defined with:

$$a_{c_j}^{(1)}(t) = a_{out_j}^{(1)}(t)h^{(1)}(S_{c_j}^{(1)}(t)),$$

$j \in \{1, \ldots, d_1\}$, $t \in \{1, \ldots, T_1\}$. Where, $h^{(1)}$ is a differentiable function.

The last output of this first hidden layer $\boldsymbol{a}_c^{(1)}(T_1)$is the encoded representation, and we repeat this representation to adapt the network to the number of time steps of the target. More precisely, $\boldsymbol{a}_c^{(1)}(T_1)$ is used as input for each time steps t, $t = 1, \ldots, T_2$, of the second LSTM hidden layer. Then, we can compute the forward pass of the second hidden layer.

For any $m \in \{c, in, out, \varphi\}$, and $t \in \{1, \ldots, T_1\}$, the net inputs at time t of the first layer are computed with:

$$\boldsymbol{net}_m^{(2)}(t) = \boldsymbol{W}_m^{(2)} \boldsymbol{a}_c^{(1)}(T_1) + \boldsymbol{U}_m^{(2)} \boldsymbol{a}_c^{(2)}(t-1),$$

where, $\boldsymbol{W}_m^{(2)} \in \mathbb{R}^{d_2 \times d_1}$ are the input weight matrices, and $\boldsymbol{U}_m^{(2)} \in \mathbb{R}^{d_2 \times d_2}$ are the recurrent weight matrices.

The activation is computed with:

$$a_{m_j}^{(2)}(t) = f_m^{(2)}(net_{m_j}^{(2)}(t)),$$

$j \in \{1, \ldots, d_2\}$, $m \in \{in, out, \varphi\}$, $t \in \{1, \ldots, T_2\}$. Where, $f_{in}^{(2)}$, $f_{out}^{(2)}$, $f_{\varphi}^{(2)}$ are respectively the activation functions of the input, output and forget gates.

We compute the internal state of each block as:

$$S_{c_j}^{(2)}(0) = 0;\ S_{c_j}^{(2)}(t) = a_{\varphi_j}^{(2)}(t) S_{c_j}^{(2)}(t-1) + a_{in_j}^{(2)}(t) g^{(2)}(net_{c_j}^{(2)}(t)),$$

$j \in \{1, \ldots, d_2\}$,$t \in \{1, \ldots, T_2\}$. Where, $g^{(2)}$ is a differentiable function.

The memory blocks outputs are defined with:

$$a_{c_j}^{(2)}(t) = a_{out_j}^{(2)}(t) h^{(2)}(S_{c_j}^{(2)}(t)),$$

$j \in \{1, \ldots, d_2\}$,$t \in \{1, \ldots, T_2\}$. Where, $h^{(2)}$ is a differentiable function.

Finally, we compute the output time series with

$$\hat{\boldsymbol{y}}(t) = \boldsymbol{W}^{(3)} \boldsymbol{a}_c^{(2)}(t),\ t = 1, \ldots, T_2,$$

where $\boldsymbol{W}^{(3)} \in \mathbb{R}^{p_2 \times d_2}$ is the matrix of output weights.

Bibliography

[1] Bayer, Justin, Daan Wierstra, Julian Togelius and Jürgen Schmidhuber. Evolving memory cell structures for sequence learning. *In :* Alippi, Cesare, Marios Polycarpou, Christos Panayiotou and Georgios Ellinas, eds, *Artificial Neural Networks – ICANN 2009*, Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 755–764.

[2] Bengio, Y., P. Simard and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*. March 1994, 5(2), p. 157–166.

[3] Bishop, Christopher M. Novelty detection and neural network validation. *IEE Proceedings-Vision, Image and Signal processing*. 1994, 141(4), p. 217–222.

[4] Bishop, Christopher M *and al. Neural networks for pattern recognition*. Oxford university press, 1995.

[5] Bontemps, Loïc, Van Loi Cao, James Mcdermott and Nhien-An Le-Khac. Collective anomaly detection based on long short-term memory recurrent neural networks. 11 2016, p. 141–152.

[6] Boyd, Stephen and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[7] Cevikalp, Hakan and Bill Triggs. Efficient object detection using cascades of nearest convex model classifiers. *In : Proceedings / CVPR, IEEE Computer Society Conference on Computer Vision and Pattern Recognition.*, 06 2012, p. 3138–3145.

[8] Chang, Wei-Cheng, Ching-Pei Lee and Chih-Jen Lin. A revisit to support vector data description. 2015.

[9] Cho, Kyunghyun, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. *In : Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar : Association for Computational Linguistics, October 2014, p. 1724–1734.

[10] Chung, Junyoung, Caglar Gulcehre, Kyunghyun Cho and Yoshua Bengio. Gated feedback recurrent neural networks. *In : Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, JMLR.org, 2015, p. 2067–2075.

[11] Ergen, Tolga, Ali Mirza and Suleyman Kozat. Unsupervised and semi-supervised anomaly detection with lstm neural networks. *IEEE Transactions on Neural Networks and Learning Systems*. 10 2017, PP.

[12] Gers, Felix, Jürgen Schmidhuber and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*. 10 2000, 12, p. 2451–71.

[13] Graves, A. and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. *In : Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., vol.4*, 2005, p. 2047–2052 vol. 4.

[14] Greff, K., R. K. Srivastava, J. Koutnik, B. R. Steunebrink and J. Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*. Oct 2017, 28(10), p. 2222–2232.

[15] Hobijn, Bart, Philip Hans Franses and Marius Ooms. Generalizations of the kpss-test for stationarity. *Statistica Neerlandica*. 2004, 58(4), p. 483–502.

[16] Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber *and al.* Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[17] Hochreiter, Sepp and Jürgen Schmidhuber. Long short-term memory. *Neural computation*. 12 1997, 9, p. 1735–80.

[18] Hua, Xiaopeng and Shifei Ding. Incremental learning algorithm for support vector data description. *JSW*. 2011, 6(7), p. 1166–1173.

[19] Kingma, Diederik and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*. 12 2014.

[20] Krizhevsky, Alex, Ilya Sutskever and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*. 01 2012, 25.

[21] Lee, Sang-Woong, Jooyoung Park and Seong-Whan Lee. Low resolution face recognition based on support vector data description. *Pattern Recognition*. 09 2006, 39, p. 1809–1812.

[22] Li, Shuai, Wanqing Li, Chris Cook, Ce Zhu and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. *In : Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR 2018, 2018, p. 5457–5466.

[23] Liu, Yi-Hung, Yan-Chen Liu and Yen-Jen Chen. Fast support vector data descriptions for novelty detection. *IEEE Transactions on Neural Networks*. 2010, 21(8), p. 1296–1313.

[24] Malhotra, Pankaj, Anusha Ramakrishnan, Gaurangi Anand, Lovekesh Vig, Puneet Agarwal and Gautam Shroff. Lstm-based encoder-decoder for multi-sensor anomaly detection. 07 2016.

[25] Malhotra, Pankaj, Lovekesh Vig, Gautam Shroff and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. 04 2015.

[26] Martens, James and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. *In : Proceedings of the 28th International Conference on International Conference on Machine Learning*, Madison, WI, USA : Omnipress, 2011, p. 1033–1040.

[27] Masters, Dominic and Carlo Luschi. Revisiting small batch training for deep neural networks. 04 2018.

[28] Parra, Lucas, Gustavo Deco and Stefan Miesbach. Statistical independence and novelty detection with information preserving nonlinear maps. *Neural Computation*. 02 1997, 8.

[29] Pauwels, Eric J and Onkar Ambekar. One class classification for anomaly detection: Support vector data description revisited. *In : Industrial Conference on Data Mining*, Springer, 2011, p. 25–39.

[30] Pérez-Ortiz, Juan Antonio, Felix A Gers, Douglas Eck and Jürgen Schmidhuber. Kalman filters improve lstm network performance in problems unsolvable by traditional recurrent nets. *Neural Networks*. 2003, 16(2), p. 241–250.

[31] Ranjan, Chitta, Markku Mustonen, Kamran Paynabar and Karim Pourak. Dataset: Rare event classification in multivariate time series, 09 2018.

[32] Rekha, AG, Mohammed Shahid Abdulla and S Asharaf. Lightly trained support vector data description for novelty detection. *Expert Systems with Applications*. 2017, 85, p. 25–32.

[33] Ruder, Sebastian. An overview of gradient descent optimization algorithms. *ArXiv*. 2016, abs/1609.04747.

[34] Sak, Hasim, Andrew W. Senior and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *In : INTERSPEECH*, 2014.

[35] Scardapane, Simone, Danilo Comminiello, Amir Hussain and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*. 06 2017, 241, p. 81–89.

[36] Schölkopfü, Bernhard, Robert C Williamson, Alex Smolaü and John Shawe-Taylorý. Sv estimation of a distribution s support. *Advances in Neural Information Processing Systems*. 1999.

[37] Schuster, Mike and Kuldip Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*. 12 1997, 45, p. 2673 – 2681.

[38] Schölkopf, Bernhard, Robert Williamson, Alex Smola, John Shawe-Taylor and John Platt. Support vector method for novelty detection. 01 1999, p. 582–588.

[39] Sutskever, Ilya, Oriol Vinyals and Quoc Le. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*. 09 2014, 4.

[40] Tax, David M. J. and Robert P. W. Duin. Feature scaling in support vector data descriptions. 2000.

[41] Tax, David M.J. and Robert P.W. Duin. Support vector data description. *Machine Learning*. Jan 2004, 54(1), p. 45–66. Available via the URL `<https://doi.org/10.1023/B:MACH.0000008084.60811.49>`.

[42] Wehenkel, Louis and Pierre Geurts. Applied inductive learning - lecture 5 (deep) neural networks. `http://www.montefiore.ulg.ac.be/~lwh/AIA/deep-neural-nets-29-12-2018.pdf`, 2018. Lecture slides. Accessed 04/29/2020.

[43] Williams, Ronald J and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*. 1990, 2(4), p. 490–501.

[44] Williams, Ronald J and David Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection science*. 1989, 1(1), p. 87–111.

[45] Williams, Ronald J. and David Zipser. Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity. *In : Backpropagation: Theory, Architectures, and Applications*, USA : L. Erlbaum Associates Inc., 1995, p. 433–486.

[46] Wilson, D. and Tony Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks : the official journal of the International Neural Network Society*. 01 2004, 16, p. 1429–51.