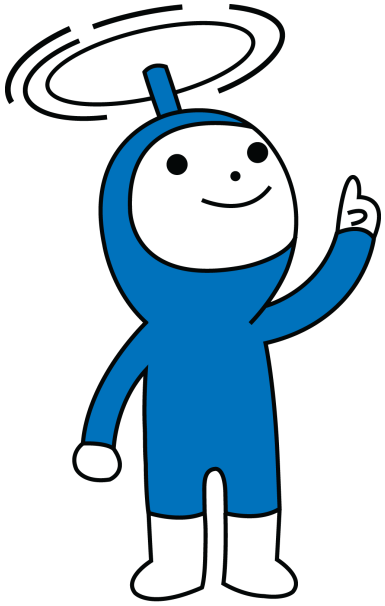
CHALMERS
UNIVERSITY OF TECHNOLOGY



Automatic scaling of machine resources in complex computing systems

Master's thesis in Master Programmes:
Engineering Mathematics and Computational Science
Complex Adaptive Systems

SIMON STRANDBERG
SIMON WESTLUND

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018:NN

Automatic scaling of machine resources in complex computing systems

SIMON STRANDBERG
SIMON WESTLUND

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2018

Automatic scaling of machine resources in complex computing systems
SIMON STRANDBERG SIMON WESTLUND

Supervisor: Staffan Truvé, Recorded Future
Supervisor: Ulf Månsson, Recorded Future
Examiner: Mats Granath, Department of Physics

Master's Thesis 2018:NN
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Marty, Recorded Future's mascot, sensing for future information.

Typeset in LaTeX
LaTeX-template adapted from Frisk (2015)
Gothenburg, Sweden 2018

Automatic scaling of machine resources in complex computing systems
SIMON STRANDBERG
SIMON WESTLUND
Department of Physics
Chalmers University of Technology

Abstract

This thesis examines the feasibility of detecting future queues in complex computing pipelines using historic time series data as training data for a recurrent neural network. It is suspected that surges of information that will be processed at different stages in the system spread and affect other processes. By predicting how large the queues are going to be a few minutes in the future, preemptive measures can be taken in order to mitigate the spikes in workload. This can be done by scaling the computing power at every node accordingly ahead of time. In order to find the useful information patterns in a very large feature space, different feature selection methods are tried and evaluated. It is found that choosing features based on their relevance to the target feature performs better than choosing features that span the feature space. Three different ways of looking at the results are tested: Naive prediction of future queue sizes, WTTE-RNN and the Sliding Box model. It is found that some predictive power exists in the former two, while the Sliding Box model performs poorly, but more tuning and data collection is needed before putting the results into production.

Keywords: Automatic scaling, Computing systems, Time series, Recurrent neural networks, LSTM, WTTE-RNN.

Acknowledgements

There are several people without whom the work with this thesis would have been a much less pleasant experience. Firstly we extend our deepest gratitude to all people, both at Recorded Future and beyond, that have lent us their time, experience, patience and expertise, helping us understand the problems at hand and working out potential solutions. There are far too many names to mention everyone. You know who you are. A special thanks of course to our supervisors, Staffan and Ulf, for proposing the project from the beginning, and helping us with guiding, ideas and inspiration from start to end. Lastly, to whomever placed a ping-pong table in the corner kitchen room: you are a catalyst for the solution of many of our problems, and a lifesaver by keeping our work ethic up after long days of reading technical reports. Thank you!

Simon Strandberg and Simon Westlund, Gothenburg, May 2018

Contents

List of Figures

List of Tables

1
Introduction

Various computing systems are met with fluctuating demand for computing power at different points in time. The effect and severity of having too little computing power differs depending on the application. It can mean missing the opportunity to perform important computations, slow response times to the user of the system, or something else. However, having less computing capability than needed is never desirable.

One way of dealing with the problem of insufficient computing power is horizontal scaling, i.e adding more machines working on the task at hand. By distributing the workload on a greater number of machines the spikes in workload and the possible following queues being built up can be mitigated. Many systems are built such that scaling certain components in the system, independently of all other components, is possible.

The question of when to scale up the number of machines can be approached by simple thresholding. When the workload, or some measurable effect of the workload, rises over a certain limit more machines are added to the specific task at hand. There are two main drawbacks with this though. Firstly, the spike has already started building up by the point at which the command to scale up and add more machines is given. Secondly, there is often some delay in conjunction with adding a new machine to the task at hand. Thus, knowing beforehand when to scale up is advantageous.

Lots of previous work on detecting spikes in cloud computing, for example by Ibidunmoye and Elmroth [2], has focused on detecting whether the current load does indeed make up a ”spike” or not. The systems studied there are, however, limited to handling a large incoming amount of requests from some external source. No previous information is contained in the system giving information on future work load. The goal is then to correctly classify if the current increase in workload is indeed a spike or not, based on the characteristics of the change of the workload. If the workload increases in such a way that a larger spike is to be expected, some measure is taken in order to deal with the increased load.

However, given that one studies a system with multiple components that are processing tasks sequentially, or doing tasks in parallel that are communicating with the same database, there may be information to be had before the spike starts to build up at the specific process being studied. One can imagine such a system as a set

of pipelines, interconnected in some more or less complicated way, with information flowing through them. Given the state of the pipes and amount of information at specific points, the rate of the flow of information and various other system metrics it will likely be possible to use current information in order to predict future behaviour.

This thesis deals with such a system.

1.1 On the nature of the task and the purpose of the thesis

Recorded Future, a world leading company in digital threat intelligence based in Gothenburg, has a system very much like the one described above. At one end of the system data is collected from multiple sources on the internet. These data are then processed in various steps, finding interesting entities and references. The collected information is then stored in a database and indexed, trying to connect bits of information and detecting patterns indicative of malicious behaviour. During the processing different processes communicate with and retrieve/store information in a set of databases. To every process there is one or multiple queues with messages yet to be handled.

On the other end of the system are the customers, getting alerts on possible exploits, leaks, IP-addresses et cetera, or asking queries to the databases that might be of interest to them in their work with digital security. These queries might be of differing size and complexity.

The purpose of this thesis then is to examine the possibility of predicting future queues building up using machine learning so that the necessary actions can be taken in order to minimise the queues or prevent the queues from happening altogether. To achieve this, data about the current and previous states of the system is used. Specifically, data about the sizes of the queues at every time, and the rate of flow of information in and out of the queues.

The historical data about the system is stored as time series in one-minute intervals. There are several different metrics for every queue, including, but not limited to: number of messages in the queue, average ingress rate, average egress rate, consumers (cores in a computer) working on the messages in the queue, et cetera. Similarly, there are a number of different metrics for every process, as well as metrics for every machine in the database clusters.

Given the complexity of the system and the interactions between all its parts, using neural networks to predict queues is a reasonable route to go. Specifically, recurrent neural networks will be used in order to utilise the sequentiality of the data, taking into consideration changes over time in the state, not just momentary snapshots in time.

In all machine learning tasks data preparation is an important task. This project will delve somewhat into feature selection given the very large possible feature space. Taking all metrics available for all processes, queues and databases would yield about 3000 features in total. With a look-back period of x minutes for every feature, the possible number of combinations for the network to learn and recognise becomes extremely large. Thus, selecting a smaller number of important features is of interest in order to enhance the signal and not train the network on what might be mostly noise.

1.2 Outline

In chapter 2, the theory behind four different feature selection methods specific for time series data are presented.

Chapter 3 then introduces the concept of artificial neural networks and their motivation, and then extends that to recurrent neural networks in general, and the LSTM cells specifically. This then constitutes the general framework for all learning methods used in the thesis. The chapter ends with the setup of the task and data for the first method tried, and presents a score system to evaluate and compare different models to each other.

In chapter 4 the concept of Weibull Time To Event Recurrent Neural Networks (WTTE-RNN) are laid out [3]. Its theory is briefly discussed and some general intuition is given. Chapter 5 then quickly presents the third method tried, the Sliding Box model for prediction of events.

The results of simulations and test are then presented in chapter 6, and chapter 7 concludes with some discussion on the results, possible improvements and future work.

2

Feature Selection

When working with machine learning, having very high dimensional data is not necessarily a problem, but definitely something that has to be dealt with. Richard Bellman coined the term "the curse of dimensionality" in connection with working on exhaustive enumeration of product spaces for optimisation problems [4]. As the number of dimensions grow, the number of possible combinations increases exponentially. Consider a Cartesian grid with spacing 1/10 on the unit cube in p dimensions. The number of points on the grid is then 10^p. Exhaustive search then quickly becomes very computationally expensive with growing dimensionality.

When working with artificial neural networks the problem is similar in nature. An artificial neural network is tuned in order to map a previously unseen input to a value or label that is close to some formerly known similar training input. If the network has not had the chance to train on something similar to the new input, the result will not be a very informed guess. What similarity is is not a trivial question, but for many purposes euclidean distance is a good proxy. In a high dimensional space most things become very far apart, and thus nothing is similar to anything. Hence the need for very large amounts of training data.

Another potential problem with including many features in a machine learning model is the possibility of training on noise, so that the results are not generalisable to new data, or not finding "the signal" in the data, so that no meaningful learning is possible at all. Thus, reducing the number of input features is important.

Feature selection methods are generally divided into three broad classes: *Wrapper*, *embedded* and *filter* methods. A wrapper method typically considers the feature selection as a search problem in which it tries some subset of the features and evaluates it using a predictive model. Features are then added or discarded depending on how important they are deemed for the result. Recursive Feature Elimination (RFE) is a typical wrapper method in which a model is built using all features, and the least important one is eliminated until the desired number of features is reached. A drawback of wrapper methods is their large demand for computational time and power, especially when starting with a very large feature space.

Embedded methods use simultaneous training and selection. A common class of embedded methods are regularisation methods that introduce certain penalties on including many features in for example a regression model so that the model becomes biased to having fewer features.

The last class then, to which all of the following proposed methods belong, is filtering methods. This class of methods use some statistical test on the data, filtering out features before any training is performed at all. Given a large feature space to begin with, filtering methods are the most computationally viable ones to use, especially in comparison with wrapper methods [5].

In the following sections a number of different filtering techniques are presented. Some of them are completely data-driven, meaning that no human intervention and possible following bias is introduced. Others require some intervention in choosing the number of features to include.

2.1 Naive selection using cross-correlation

Cross-correlation is a measure of the similarity between two series x and y at a certain offset τ defined as

$$R_{xy}(\tau) := (x * y)(\tau) = \int_{-\infty}^{\infty} x^*(t)y(t-\tau)dt \tag{2.1}$$

where $x^*(t)$ denotes the complex conjugate of $x(t)$.

In order to select features, the cross-correlation between the dependant feature and all other features is calculated. The importance of the feature is measured by how high the maximum cross-correlation is. Using knowledge about the nature of the data studied a maximum lag of T can be considered for increased computational efficiency. If it is known that data outside some time-lag τ but within a certain threshold $0 < \tau \leq T$ should not have any significant effect on the outcome, then those cross-correlations can be omitted from calculation. Given that the nature of the data is such that all features are strictly non-negative at all times, the idea is that there should be some activity in the feature prior to the examined feature rising, giving rise to a higher cross-correlation for that time lag.

Since the value of 2.1 is dependant on the size of the respective series, it is often convenient to scale the value according to

$$R_{xy}^{scaled}(\tau) := \frac{1}{\sqrt{R_{xx}(0)R_{yy}(0)}} R_{xy}(\tau) \tag{2.2}$$

so that the value of $R_{xx}^{scaled}(0) = 1$ and $R_{xy}^{scaled}(\tau) \in [-1, 1]$. After normalising the cross-correlations according to 2.2 and calculating $R^{max} := \max_{0 \leq \tau \leq \tau_{max}} R_{xy}^{scaled}(\tau)$, the n features with largest R^{max} are selected.
One possible drawback of naively choosing all the highest correlating features however is the possibility of including redundant information. If multiple features have a high cross-correlation with the target feature at some point in time, they should also have high cross-correlation with each other, thus sharing information, and only one of them could be enough.

2.2 Iterative Cross-correlation Filtering

Iterative Cross-correlation filtering, ICF for short, uses the principle of cross-correlation to perform a more advanced forward selection of features excluding features that are pairwise correlated, or just noise. The algorithm is performed in multiple steps, and the proceedings and selection rules demand a rather lengthy explanation. The interested reader can find the exact proceedings and motivations in [6]. This section will provide only a brief summary in order to give some intuition to the method.

As described in [6], ICF works on the following three intuitions:

- Select a variable to be included if it is not correlated with any other variable.
- Eliminate variables that are correlated with all other already selected variables.
- Act conservatively; if the rules for selection or elimination result in a working set of mutually correlated variables, maintain only the features that are less correlated with those selected.

The first part of the algorithm calculates a *redundancy matrix* $R \in \{0,1\}^{D\times D}$, where D is the total number of features. Every entry R_{ij} is 1 if feature i and j at some point in time have a cross-correlation close to 1, or zero otherwise. Features awarded with a 1 are said to be pairwise redundant. That is, they contain the same information, but with some time-lag. The diagonal of R is set to 0 in order to rule out the trivial cross-correlations of a series with itself at no lag.

Secondly, features that are likely noise are deleted. This is determined by checking the *auto-correlation* (cross-correlation of a series with itself). If the average auto-correlation of all non-zero lags is close to 0, then the feature is deemed noise and is therefore deleted.

The last step of ICF is to assign all remaining features in the set $\mathcal{F}$ of unassigned features to either the group of selected features $\mathcal{SF}$, or the set of deleted features $\mathcal{DF}$, by consecutively checking four selection rules.

1. If row $R_{i\cdot}$ is completely uncorrelated with the others ($R_{i\cdot}$ contains only zeros), then that feature is removed from $\mathcal{F}$, added to $\mathcal{SF}$, and all corresponding entries in R are removed. If this in turn creates a new feature that is completely uncorrelated with all others, that feature is assigned to $\mathcal{DF}$.
2. If row $R_{i\cdot}$ is correlated with all others and there is at least 1 non-completely correlated feature (i.e., R does not contain only one off-diagonal), then add i to $\mathcal{DF}$ and remove it from $\mathcal{F}$ along with all corresponding entries in R.
3. If all remaining features in $\mathcal{F}$ are correlated with each other, select the one that is least correlated with the features already in $\mathcal{SF}$ and add that feature to $\mathcal{SF}$. Move all others to $\mathcal{DF}$ and then terminate the algorithm.
4. If none of the earlier rules apply, then find the remaining feature i least correlated with all others in $\mathcal{F}$. Define $S(i) \in F$ to be the set of all features correlated with i and chose $j \in S(i)$ that is most correlated with the features in $\mathcal{SF}$. Lastly, add i to $\mathcal{SF}$, j to $\mathcal{DF}$ and remove them both from $\mathcal{F}$ along with all corresponding entries in R.

2.3 Granger causality

The concept of Granger causality was originally proposed as a means of identifying causal interactions between different time-series in econometry [7]. Recent work [8] has used the concept of Granger causality in order to design a feature selection algorithm for multivariate time-series.

The basic principle of Granger causality is a statistical hypothesis test, comparing if a variable x increases the predictive power on y compared to an ordinary auto-regressive model. More formally, let x and y be two different time series, let x_t and y_t be their most recent value and let x_{t-k} and y_{t-k} be their lagged values with a lag of k time steps. First conduct the two regressions:

$$\hat{y}_{t1} = \sum_{k=1}^{l} a_k y_{t-k} + \epsilon_t \tag{2.3}$$

$$\hat{y}_{t2} = \sum_{k=1}^{l} a_k y_{t-k} + \sum_{k=1}^{w} b_k x_{t-k} + \eta_t \tag{2.4}$$

where $\hat{y}_{t_i}$ is the least square regression model fitted according to the equations, a_k and b_k are regression coefficients, l and w are the maximum allowed lag of y and x respectively and ϵ_t and η_t are the prediction errors. The size of w and l could in theory be infinite, but are assumed much shorter and are usually determined by the Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC) [8].

Variable x is then said to Granger cause y if and only if 2.4 improves the accuracy as compared to 2.3 by some statistical test, for example the F-test, with some specified level of certainty p.

The decision to include feature $x^{(i)}$ in the model is made if $x^{(i)}$ Granger causes y but y does not Granger cause $x^{(i)}$. This is then repeated for all features.

2.4 CLeVer based methods

CLe**V**er, first proposed in 2005, is short for *descriptive **C**ommon principal component **L**oading based **V**ariable subset selection method* [9]. ***CL**e**V**er* is actually a collection of three different, but theoretically and application-wise similar, methods. The basis for them is Principal Component Analysis (PCA) on separate Multivariate Time-Series (MTS) items.

***CL**e**V**er* was suggested with a set of different MTS's in mind, such as one would get when measuring values on a set of different people - for example doing MRI scans on 10 different patients - thus generating multiple disjoint MTS's but with similar characteristics. This thesis works with one very long continuous MTS over 3 months, but that can easily be cut into several shorter MTS's.

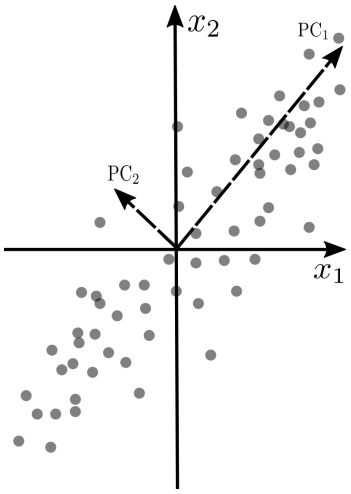


Figure 2.1: Example of principal components in data. Most of the variance in the data is along the first principal component.

What ***CLeVer*** does not do is take into account lagged values of a feature, or automatically choose the number of features to include, thus requiring some human intervention.

In order to understand ***CLeVer***, a brief recollection on PCA is first presented, including Common PCA, and then the three different ***CLeVer*** methods. Unless otherwise stated, the following theory is taken from [9].

2.4.1 Principal Component Analysis

Intuitively, the purpose of PCA is finding the directions in the data with the maximum variance, or rather the directions which minimises the variance of the residual errors when the data is projected there. In most data mining settings the aim is to find a lower dimensional space to which the data can be projected while losing as little useful information as possible. Figure 2.1 shows an example of a two-dimensional set of data where the data is spread mainly along one axis, the first principal component.

This is done by performing a Singular Value Decomposition (SVD) on the covariance matrix corresponding to the data set so that

$$A = V\Sigma U^*$$

where A is the covariance matrix, V contains the loadings for the eigenvectors spanning the principal components and Σ is a square matrix with the corresponding eigenvalues, where the size of the eigenvalue corresponds to the variance along that direction.

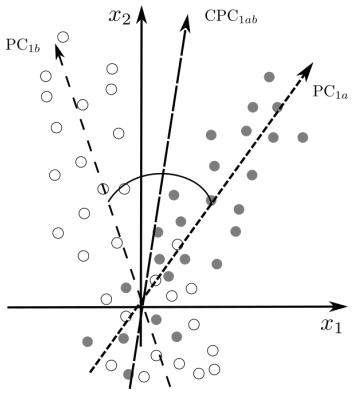


Figure 2.2: Example of Common Principal Components between two data sets a and b with their respective leading principal components PC_{1a} and PC_{1b}.

2.4.1.1 Common Principal Components

When having several sets of MTS's the PC's can be calculated for each one. One method to then calculate a set of Common Principal Components (CPC) was proposed in [10]. The concept used is to bisect the angles between the corresponding PC's from the various MTS's. Suppose every MTS is described by its leading p principal components. The ith CPC ($i = 1, ..., p$) is then obtained by successively bisecting the angles between the ith PC in every MTS. An example is shown in figure 2.2 where the first CPC is found by bisecting the angle between the leading PC's of the two sets of data.

The CPC's then describe the subspace that most closely agrees with all the subspaces from the collected MTS's. Constructed in this way it is also ensured that the CPC's are orthogonal, and ordered non-increasingly. Furthermore the value of the jth component in every CPC is corresponding to the contribution from the jth feature in the original feature space, just as would be expected from ordinary PC's.

All of the three following methods start off with a set of CPC's calculated from a set of MTS's as described above.

2.4.2 CLeVer-*Rank*

CL*e****V****er -Rank* chooses features based on their contribution to the CPC's, according to the L_2-norm of the corresponding loadings from that feature. For variable v_i the ith loading l_i of every CPC is used to assign a score according to

$$|v_i| = \sqrt{l_{1,i}^2 + ... + l_{p,i}^2} \tag{2.5}$$

The intuition here is that variables with a high score has contributed a lot to at least one and probably several of the CPC's. Then the K highest ranked variables are chosen to be included in the model.

One drawback of this scheme is that it is prone to selecting redundant variables. It is possible that an even smaller amount of variables might have equal predictive performance, since many high ranked variables probably share information through correlation with each other. It is also possible to miss important variables that get a low ranking, but are only important when paired with certain other variables. ***CL**e**V**er -Cluster* aims to solve these problems to some extent.

2.4.3 CLeVer-*Cluster*

Instead of choosing variables that are individually important, ***CL**e**V**er -Cluster* tries to select variables that hold different kinds of information and complement each other. By performing a K-means clustering of the CPC's it is possible to find the variable closest to each cluster centre and select that to be included in the model; The idea being that variables that are highly correlated and thus placed in the same cluster will share information that are redundant when used together with all variables in the same cluster.

To get better performance, the K-means clustering is performed 20 times and then the iteration with the least within cluster sum of Euclidean distances to the cluster centroid is chosen. This increases the possibility of getting more clearly defined, tight, clusters.

Whilst ***CL**e**V**er -Rank* always returns the chosen number K of variables, ***CL**e**V**er -Cluster* returns at most K, being the number of clusters formed. If several clusters have the same feature closest to the cluster centre, then fewer features will be chosen.

2.4.4 CLeVer-*Hybrid*

***CL**e**V**er -Hybrid* utilises the strengths of both previous methods. Firstly the clustering is performed according to the same method as in ***CL**e**V**er -Clustering*. Then the variables within the cluster are ranked in the same way as in ***CL**e**V**er -Rank*. Instead of choosing the feature closest to the cluster centroid, the highest ranking feature for every cluster is chosen instead.

When referring to the ***CL**e**V**er* method when doing feature selection later in this thesis, ***CL**e**V**er -Hybrid* is the one used.

3
Recurrent Neural Networks

An artificial neural network is a model used for machine learning inspired by the human brain, trying to mimic how neurons interact with each other through electrical signals being sent through synapses. Depending on the strength of the connection, neurons affect each other to varying degrees. The artificial neural network is trained by being fed data and then optimising an energy function, in the hope that it learns to recognise patterns in the data and is able to correctly label or assign an adequate value to a new, previously unseen, set of data.

In this chapter the theory behind basic feed-forward neural networks will be explained and formalised in section 3.1, whereupon the extensions to recurrent neural networks will be made in section 3.2. Readers familiar with the basic theory can skip directly to section 3.3, where a framework for doing the predictive regression on future queue sizes specific for this thesis is set up.

3.1 Feed-forward neural networks

The performance of neural networks can be outstanding in finding non-linear and complex relationships between inputs and outputs. They also generalise very well, using their lessons learned from training data on previously unseen data efficiently [11]. There are several distinct network architectures that are used for different types of tasks, a common one being the feed-forward neural network, which consists of a network of artificial neurons with unidirectional connections between each other.

3.1.1 Neuron representation

As previously mentioned neural networks are inspired by how neurons in our brain interact with each other. A neuron, as seen on the left of figure 3.1, consists among other things of a cell body, dendrites, synapses and an axon. Transmitting a signal to another neuron is a complex chemical process, but it can be simplified to releasing certain substances to the receiving cell in order to increase or decrease the electrical potential inside its body. If the potential exceeds a threshold the cell would fire a pulse along the axon, which would then affect other cells [12].

An artificial neural network is a hugely simplified version of this process. A single neuron in an artificial neural network, as can be seen on the right of figure 3.1, receives numbers from previous parts of the network as input and calculates

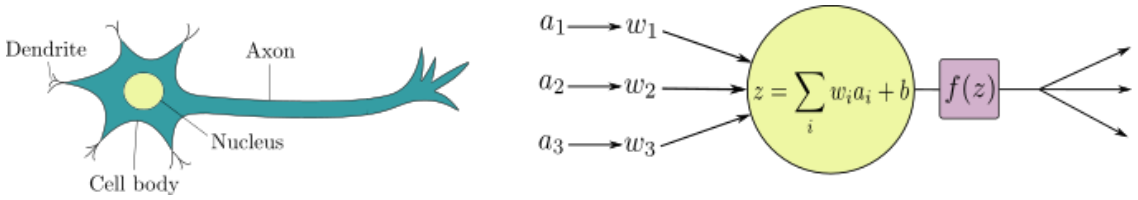


Figure 3.1: On the left is a simplified illustration of a real neuron. On the right is a neuron representation in an artificial neural network, where a_i are the inputs, w_i are the weights, b is the bias, z is the input sum and f is the activation function.

a weighted sum with an added bias value. Variables called *weights* and *biases* are used in the computation, which goes by the name *input sum* and is expressed as

$$z = \sum_i w_i a_i + b, \tag{3.1}$$

where b is the bias, w_i are the weights and a_i are the inputs. The weights and biases are tweaked during the training of the model in order to create the desired behaviour. The result of the input sum will consequently be passed through an activation function to determine what should be advanced in the network [12].

3.1.2 Activation function

The neuron does not take any bounds of its output value into consideration, hence it can be any real number. In order to know when the neuron should fire the value needs to be passed through an activation function. An activation function can be as simple as a step function, where the neuron would fire if the output exceeds a certain threshold. However, a lot of information is lost when using a binary activation function like the step function.

If instead the activation function was linear, no information about what the neuron wanted to output is lost. Although there is an essential problem with this as well. If the output of all neurons is calculated by a linear function and then activated by another linear function the output of the network of neurons would just be a linear function of the inputs. Hence, the ability to stack layers of neurons to calculate complex non-linearities is lost. In most cases where neural networks are effective they need to be able to find nonlinear relationships between the input and the output [13]. Effectively, the activation function is what creates non-linearity in a neural network, provided that the activation function itself is nonlinear.

Three of the most well used activation functions for neural networks, seen in figure 3.2, are the sigmoid function, the hyperbolic tangent function and the rectified linear unit (ReLU) [14]. They have different properties that make them efficient at different tasks. The sigmoid function is better at classification tasks rather than regression tasks, while ReLU has properties that makes the network faster to train as compared to the hyperbolic tangent function for example [14].

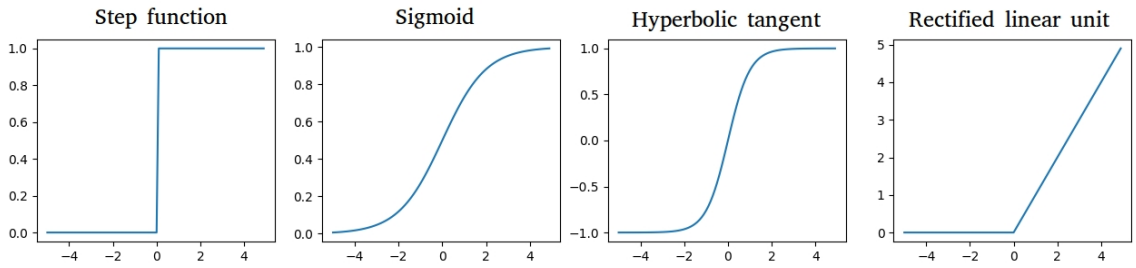


Figure 3.2: Four examples of activation functions used to limit the neuron output and determine the value to be sent forward in the network.

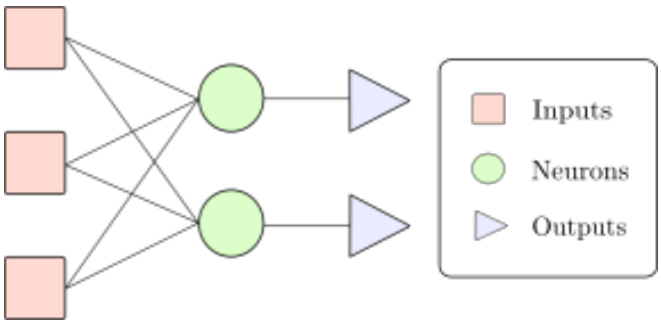


Figure 3.3: An illustration of a single layer feed-forward neural network. Each line connecting inputs to neurons has a corresponding weight and each neuron contains a bias value used to calculate the input sum.

3.1.3 Network structure

Connecting these neurons to each other creates a neural network. Figure 3.3 shows a single-layer neural network where the inputs are fed to the output that consequently provides a result. So far this is a quite simple mathematical operation to process a few input values. However, increasing the size of the network both in regards to the amount of neurons and to the amount of layers in the network can create a large network with ability to find rather complex patterns in the input data [13]. An example of a multi-layer neural network is illustrated in figure 3.4 where the inputs are first fed to what is called a hidden layer where they are processed and fed forward to the output layer.

3.1.4 Energy function

In order to train the neural network it is necessary to be able to evaluate its performance. The training process is essentially about minimising the network's energy function, or loss function as it is often called. The purpose of the loss function is to measure how well the network is performing on the given data. In a regular supervised learning problem the actual results, also called labels, of the training data are known and can be used to compare with the output from the network.

A general loss function can be expressed as

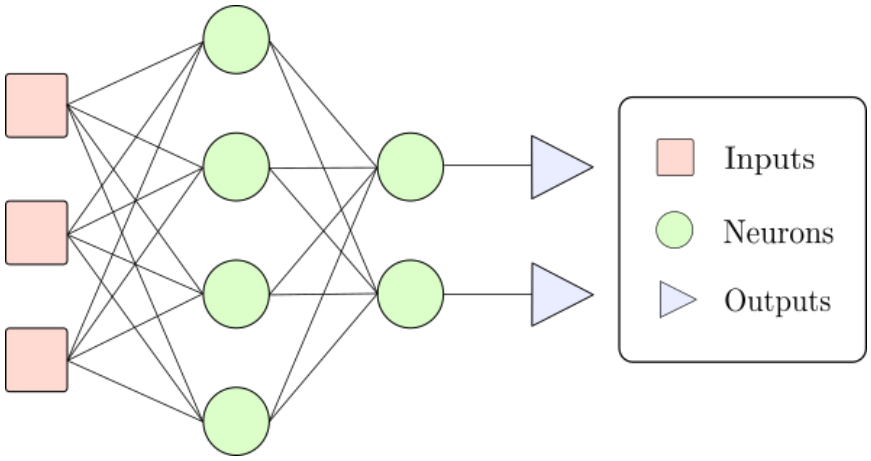


Figure 3.4: A visualisation of a multi layer feed-forward neural network. This neural network has two layers of neurons; a hidden layer and an output layer. The hidden layer consists of four neurons and the output layer has two. All connections between inputs and neurons, as well as between neurons and neurons, contain weights and each neuron also has a corresponding bias value.

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i), \tag{3.2}$$

where n is the number of training cases, L is a function of the desired outputs y_i and the predicted outputs $\hat{y}_i$. The loss function's requirements are that it needs to be averageable over training cases and it needs to be possible to write it as a function of outputs [15]. A common loss function is mean squared error of which the standard form is defined as

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{3.3}$$

where n is the number of training cases and y represents the actual results while $\hat{y}$ is the output from the network [16].

3.1.5 Backpropagation

Arguably the most important part of neural networks is their ability to learn from experience. Backpropagation is what makes that possible. It works by changing the variables of the network, i.e. the weights and the biases, according to the result of the loss function. Understanding how differences in weights and biases can reduce the loss function will provide the potential to improve the network's performance [13]. The backpropagation algorithm is based on gradient descent. By calculating the gradient of the loss function with respect to the weights and biases it is possible to minimise the error of very complex nonlinear functions [13].

In order to use gradient descent to optimise the loss function it is necessary to

know the partial derivatives of the loss function with respect to the weights and biases. These partial derivatives can be expressed as

$$\frac{\partial \mathcal{L}}{\partial w_{kj}^l} \text{ and } \frac{\partial \mathcal{L}}{\partial b_j^l} \tag{3.4}$$

for weights and biases respectively, where the superscript l indicates the layer and subscripts k and j denotes a specific neuron. An error signal of each neuron is defined as

$$\delta_k^l \equiv \frac{\partial \mathcal{L}}{\partial z_k^l} \tag{3.5}$$

in order to calculate the partial derivatives above. This simply expresses how much the error changes when the input sum is changed. This input sum is defined as

$$z_k^l = \sum_i w_{ik}^l a_i^{l-1} + b_k^l, \tag{3.6}$$

where z_k^l is the result of a_k^l being passed through the activation function. By using the chain rule the right hand side of equation 3.5 can be written as

$$\frac{\partial \mathcal{L}}{\partial z_k^l} = \frac{\partial \mathcal{L}}{\partial a_k^l}\frac{\partial a_k^l}{\partial z_k^l} = \left(\sum_m \frac{\partial \mathcal{L}}{\partial z_m^{l+1}}\frac{\partial z_m^{l+1}}{\partial a_k^l}\right)\frac{\partial a_k^l}{\partial z_k^l} = \left(\sum_m \frac{\partial \mathcal{L}}{\partial z_m^{l+1}} w_{mk}^l\right)\sigma'(z_k^l). \tag{3.7}$$

This can in turn be formulated as

$$\delta_k^l = (\sum_m \delta_m^{l+1} w_{mk}^{l+1})\sigma'(z_k^l). \tag{3.8}$$

This equation enables the ability to recursively calculate all the error signals in the network, which will be necessary in the computation of the partial derivatives in equation 3.4. The equation can perhaps be more easily understood when expressed in matrix form as

$$\delta^l = (w^{l+1})^T \delta^{l+1} \odot \sigma'(z^l), \tag{3.9}$$

where applying the transpose weight matrix can be seen as moving the error backwards in the network. Taking the component-wise multiplication of the derivative of the activation function moves the error backwards through the activation function. At this stage it would be helpful to be able to express the partial derivatives in equation 3.4 as a function of the error signals. It is easy to see that $\frac{\partial z_k^l}{\partial b_k^l} = 1$ which means that

$$\frac{\partial \mathcal{L}}{\partial b_k^l} = \frac{\partial \mathcal{L}}{\partial z_k^l}\frac{\partial z_k^l}{\partial b_k^l} = \frac{\partial \mathcal{L}}{\partial z_k^l} = \delta_k^l. \tag{3.10}$$

The same procedure with the partial derivative of the loss function with respect to the weights results in

$$\frac{\partial \mathcal{L}}{\partial w_{kj}^l} = \frac{\partial \mathcal{L}}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{kj}^l} = \delta_k^l a_j^{l-1}. \tag{3.11}$$

Now that expressions for the partial derivatives in equation 3.4 have been derived and can be computed with known quantities it is time to update the weights and biases according to these gradients. The standard updating rule using gradient descent gives

$$w_{kj}^{l} \leftarrow w_{kj}^{l} - \eta \frac{\partial \mathcal{L}}{\partial w_{kj}^{l}} \tag{3.12}$$

$$b_{k}^{l} \leftarrow b_{k}^{l} - \eta \frac{\partial \mathcal{L}}{\partial b_{k}^{l}}, \tag{3.13}$$

where η is the *learning rate*. However, using standard gradient descent is not the only way to update the variables in order to improve the network's performance. There are more efficient algorithms that tend to increase the learning speed. A few examples are *AdaGrad*, *RMSprop* and *Adam*. The difference between these methods and the standard gradient descent method is that they have adaptive learning rates [17]. They all have different ways of doing this, which results in unique properties and therefore different performance depending on the type of data.

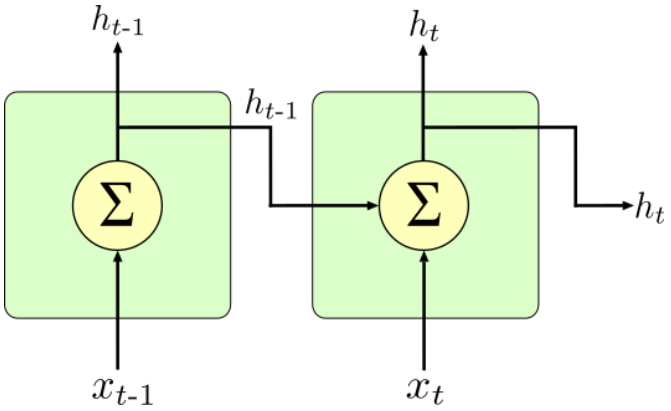


Figure 3.5: A visualisation of two RNN units and the connection between them. The hidden state of one unit is fed into the next unit along with the next input value, allowing the RNN to utilise the sequentiality in the data.

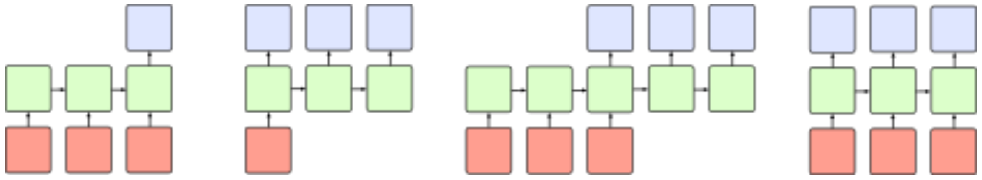


Figure 3.6: Examples of different types of RNN structures. Many inputs can give one output, one input can give many outputs and multiple inputs can give multiple outputs.

3.2 Recurrent neural networks

Many problems suited for machine learning are based on sequential data. Translating natural language, speech synthesis and time series prediction are just a few examples [18]. Ordinary feed-forward neural networks do not handle its input data as a sequence. The order of the data is not considered and thus it will not be able to use information that exists in the sequence itself. Recurrent Neural Networks (RNNs) consist of units that are not only connected to units in the next layer, but also to units in its own, which aims to solve this problem.

3.2.1 Simple RNN

The reason recurrent neural networks handle sequential data so well is that they use the hidden state of the previous time step as part of the input to the next unit. The hidden state is a function of the input and the hidden state h_t from the previous unit, such that

$$h_t = f(W \cdot x_t + U \cdot h_{t-1} + b), \tag{3.14}$$

where W is the weight matrix for the inputs, U is the weight matrix for the hidden state, b is the bias, x_t are the inputs and h_{t-1} is the hidden state from the previous unit. The function f is the activation function of the unit. The hidden state will be outputted to the following unit and sometimes also to the next layer of units. Figure 3.5 is a visualisation of this.

The unique connections in a recurrent neural network enhances the flexibility of its general structure. Figure 3.6 illustrates how the sizes of the input and output data can differ between models. This thesis will work only with the ”many-to-one” structure to the left in the figure, using the inputs from multiple sources and giving one guess after seeing the complete training history.

Aside from the addition of the hidden state's recurrent nature the RNNs are not very different from a feed-forward neural network. However, there is a slight difference in how updating the weights and biases is handled in recurrent neural networks. Instead of the usual backpropagation algorithm described in section 3.1.5 a somewhat different algorithm called backpropagation through time (BPTT) is used. The difference is that this time the loss is not only backpropagated through layers, but also through the time steps in each recurrent layer.

Trying to exploit memory by connecting units as in a regular RNN comes with some consequences. Using BPTT to train the RNN will unfold the RNN through time, essentially making it act like a deep neural network [19]. It quickly becomes difficult to train because of the vanishing or exploding gradient problem. As explained in section 3.1.5 the error signals in the network are affected by later error signals. These in turn affect the gradients when updating the weights and biases. Hence, gradients less than one will cause even smaller ones in the previous layer. In the same way, gradients larger than one will cause larger gradients in the previous

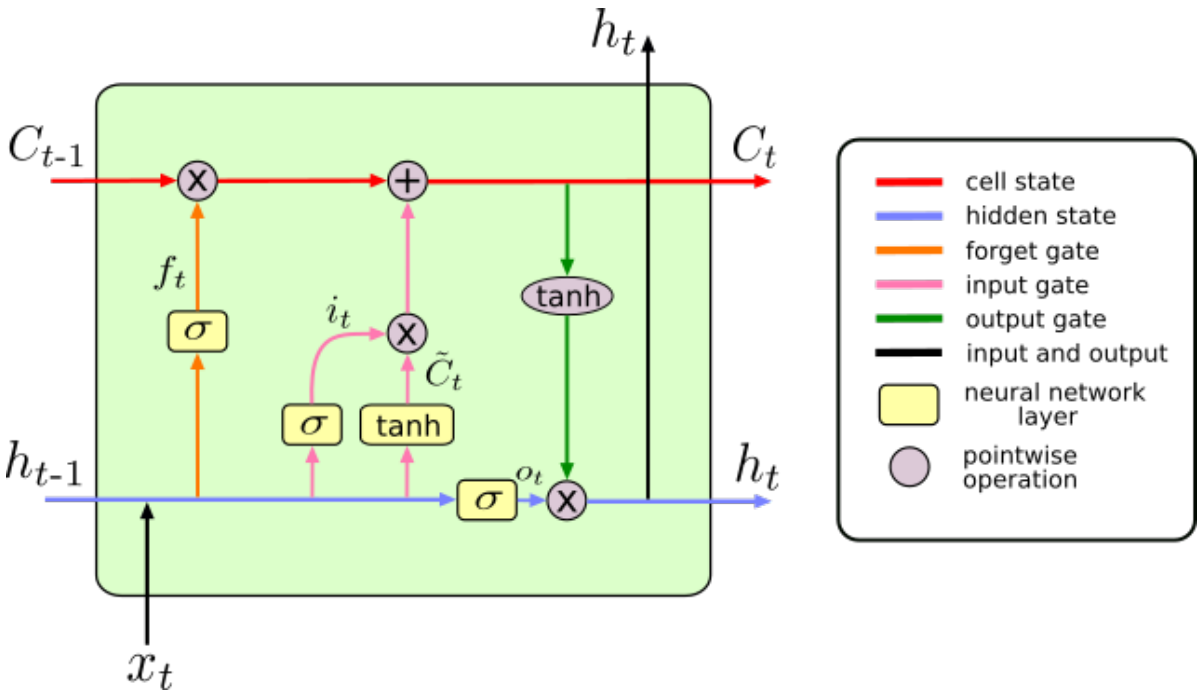


Figure 3.7: An LSTM unit and its underlying components. Both the hidden state and the cell state is sent forward in the network, unlike the RNN unit where there only is a hidden state that is kept.

layer. If the network is deep enough this will cause the gradients to either vanish or explode, making it difficult to train each layer of the network sufficiently. For example, extremely small gradients in the early layers will update those weights and biases a lot slower than the larger gradients in the network's later layers. This issue makes recurrent neural networks difficult to train as the number of time steps increase [19]. In order to deal with this problem a unit called Long Short-Term Memory (LSTM) was invented.

3.2.2 LSTM

Replacing the basic RNN unit with the more advanced LSTM unit often enhances the ability to train the network, which leads to an improved performance of the RNN [20]. The principal idea of the LSTM unit is called the cell state and it runs through each time step in the chain of units. Each LSTM unit can potentially add or remove information to the cell state. The information is optionally changed through structures called gates. There are three of these gates; the forget gate, the input gate and the output gate, each with their own purpose. The interaction between cell state, hidden state and gates is illustrated in figure 3.7. The forget gate essentially decides what information to discard from the cell state. It analyses the hidden state from the previous unit and the input to the current unit and outputs a decimal fraction for each value in the cell state to represent how much of the information to keep. The following equations describing how the LSTM unit updates its cell state and calculates its hidden state is taken from Lipton, Berkowitz and Elkan's article "A Critical Review of Recurrent Neural Networks for Sequence Learning" [21]. The

equation that corresponds to the action of the forget gate can be expressed as

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \tag{3.15}$$

where σ is the sigmoid activation function of the forget gate, W_f and b_f represent the weights and biases of the forget gate and h_{t-1} and x_t is the hidden state and the input respectively. The subsequent step is for the unit to decide what information should be added to the cell state, which is regulated by the input gate. In this process two neural network layers combine their respective contributions. The first of the layers is a sigmoid layer that decides which values to update according to the following equation

$$i_t = \sigma(W_t \cdot [h_{t-1}, x_t] + b_i). \tag{3.16}$$

The second layer creates new values, $\tilde{C}_t$, that are candidates to be added to the cell state. The equation corresponding to this layer is defined as

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C). \tag{3.17}$$

The results of these layers are then multiplied to create update values for the cell state. The complete update of the cell state in a single LSTM unit is described by the equation

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t. \tag{3.18}$$

The only remaining component of the LSTM unit to be explained is the output gate, which intends to decide the output of the unit. The output is the hidden state of the current unit and depends on the input, the hidden state of the previous unit and the updated cell state. The equations used to calculate the hidden state are defined as

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{3.19}$$

$$h_t = o_t \cdot tanh(C_t). \tag{3.20}$$

The reason that this architecture prevents the gradients from vanishing or exploding when backpropagating through time is the cell state [22]. When calculating the derivative of C_t with respect to C_{t-1} only the forget gate term remains from equation 3.18. The output of the forget gate is essentially the weights for the cell state, activated by the identity function. The derivative of the identity function is one, so if the output of the forget gate is one as well the information from the previous cell state will be kept unchanged. The network can then inherently learn what information is important to keep, without it vanishing or exploding because of the small or large gradients. Models using LSTM units typically outperform models using basic RNN units, but LSTMs are slower to train. The reason for this is evident in equations 3.15-3.19. LSTMs contain several additional weight matrices and biases that are being updated throughout the backpropagation. Hence, the many advantages of LSTMs such as the gained robustness while training and improved long term dependency come at the cost of slower training.

3.3 Predictive regression with RNN

As mentioned in sections 3.1 and 3.2, neural networks in general are proficient at finding complex non-linear relationships between input and output, while recurrent neural networks in particular are good at handling time series data. This seems like a perfect combination for the task at hand. The goal is to create a recurrent neural network and use it to predict future queue sizes in the system while feeding data about the current state of the system to the model. The queues consist of the number of documents currently waiting to be processed by a particular process in the pipeline. The inputs are features chosen with the methods described in chapter 2. Since there is so little knowledge of the relationships between the input features and the output queue size, a neural network is a viable model for solving this problem. The recurrent neural network model is coded in *Python* using *Keras* as a framework and *TensorFlow* as its back end. The model is trained on 3 months of collected historical data, sampled at one minute intervals.

3.3.1 Prepare data for supervised learning with RNN

In a supervised learning problem, such as the one present in this thesis, the aim is to train the neural network on previous events, so that similar patterns occurring in the future will be detected and outputed sufficiently close to the actual target value. To do this, the data needs to be set up in an effective way.

Each data point consists of n features, each with a value from T time steps. Each data point is also associated with its own target value, which is the size of the specific queue t_f time steps into the future. The data for a single data point looks like

$$[x_1^{(t-T)}, x_2^{(t-T)}, ..., x_n^{(t-T)}, x_1^{(t-T+1)}, ..., x_n^{(t-T+1)}, ..., x_n^{(t)}, x_{target}^{(t+t_f)}], \tag{3.21}$$

where t is the current time, x_i are the input features and x_{target} is the target feature. The model will use the first $n \cdot T$ values as input to base its prediction on, where all data from the same feature is fed to the same RNN cell sequentially. It will then evaluate its performance by comparing the output to $x_{target}^{(t+t_f)}$ through its loss function and update itself according to the result of this evaluation.

The value of T could in theory be such that all available previous history is used in classifying every new point. However, it is reasonable to believe that the long term dependencies will not be that large. Documents entering the system are usually processed and done within a couple of minutes, and only in rare and extreme cases does the delay, defined as the median time for a document entering until it is finished processing, rise above 20 minutes. Thus, T can in practice be much shorter than 3 months, being the limit of the amount of data collected. This also has the advantage of added computational efficiency and limiting abundant information. Having larger T means the complexity of the patterns rise, and with a very large feature space to begin with that is not desirable.

3.3.2 The RNN model

The basic RNN model consists of two layers of LSTM cells with sizes 60 and 30 respectively. The second LSTM layer returns a sequence which is fed into a dense multi-layer neural network. The layers are of sizes 30 and 10 nodes plus the output layer containing one node. The biggest advantage of using LSTM cells instead of vanilla RNN cells is that LSTM cells handles vanishing and exploding gradients a lot better. This means that it is possible to train the model on longer sequences. In this case the sequences are not necessarily too long for simple RNN cells to cope with. The choice of using LSTM cells instead of simple RNN cells is based on the fact that LSTM networks often outperform simple RNN networks and the only downside is that the training is a bit more time consuming [22]. The LSTM layers use the hyperbolic tangent as its activation function, while the dense layers use the rectified linear unit. The shape and size of the network was decided by intuition in combination with some quick performance testing. The model is trained to optimise the mean squared error between the prediction and labeled data using the Adam optimizer.

3.3.3 Training the model

The model is trained and evaluated on four separate data sets with features from the four different feature selection methods outlined in chapter 2. The data set is split into data points consisting of values from every feature 10 time steps back in time. Since the data sets consist of minute-interval data, this means that the previous 10 minutes of data is used in order to make the prediction of the future state. The label is set to the value of the target feature 5 minutes ahead of the last time stamp in the corresponding data point. Using 90 days worth of data on minute-interval results in 129600 data points. A negligible amount of these, 15 to be exact, cannot be used because they are at the boundaries of the data set and do not have a sufficient amount of data from previous time steps to base a prediction on. The model is trained using 80% of the data, the rest being saved for validation, for 100 epochs with a batch size of 64, meaning that it iterates through every single data point 100 times in total and updating its variables once every 64th data point.

3.3.4 Evaluating the results

The method of evaluation obviously has a direct effect on the interpretation of the result, hence it is a vital choice. The usual approach is to use the loss function for evaluation as well, but that might not be the most significant in all cases. This standard LSTM model, predicting the size of the queue t_f minutes into the future, is using *mean squared error* as a loss function. This makes sense because it punishes large deviations much more than small ones. Thus, if the prediction is a little bit off when the queue is very low it does not matter that much, but predicting low values when a peak rises up is punished more, encouraging the model to prioritise finding the occurrence or non-occurrence of peaks. However, the exact difference between the predicted and actual value along the whole curve is not the most important feature as regarding to how well the model will perform in action.

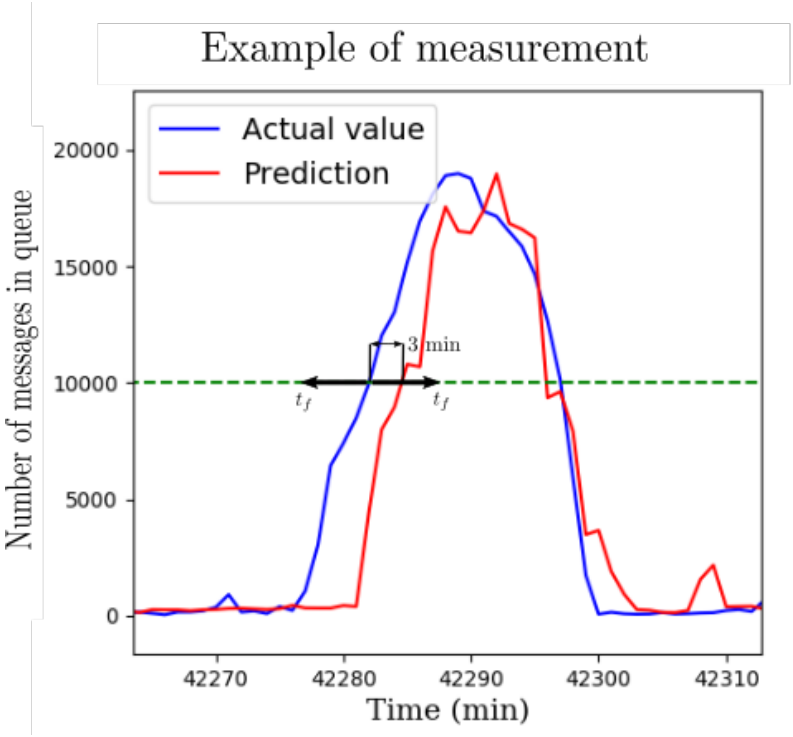


Figure 3.8: An example of a successfully predicted peak. If the prediction is within t_f away from the actual value it is considered a successful prediction. In this example the prediction is plotted to pass the threshold three minutes after the actual curve does so. However, information is still gained from this result. Since the prediction is plotted directly after making a prediction and its corresponding label from the real data is actually $t_f = 5$ minutes ahead of time, being three minutes too late in the plot actually means predicting the peak two minutes in advance.

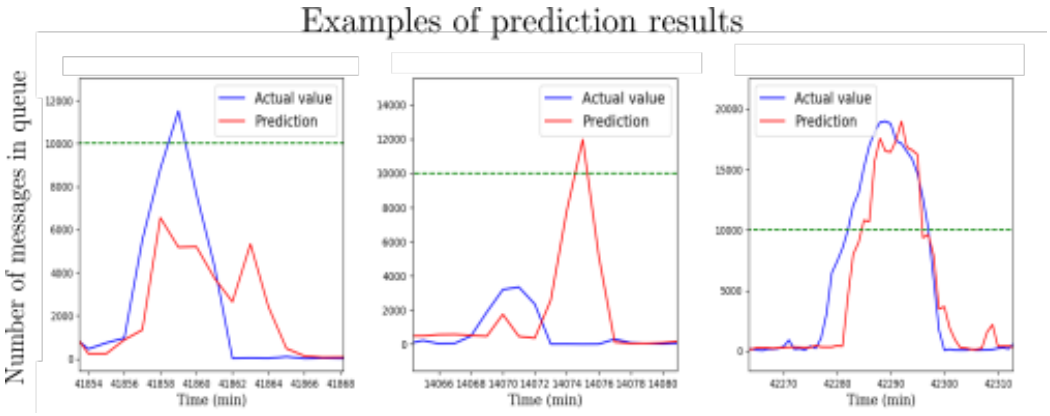


Figure 3.9: Examples of different prediction outcomes. The left panel shows an example where there is an actual peak that exceeds the threshold, but the model fails to predict it; a so called false negative. The middle panel is an example of a false positive, when the queue size never exceeds the threshold but the model predicts that it will. The right panel shows a successful prediction where the model rather accurately mimics the actual behaviour of the queue.

The aim is for the model to sufficiently predict future queue sizes in order to scale up the computing power for the corresponding process proactively, which will consequently lead to smaller queues and thereby also a decreased total processing time. The current decision for when to scale up is based on certain threshold values. When the size of a queue exceeds one of its threshold values, a scale up command is sent for its corresponding process. If the model can predict when the queue size will surpass a threshold value it can send the scale up command ahead of time, while keeping the same method for scaling up. So rather than evaluating the model by every value on the curve, it is more relevant to evaluate how well and how much ahead of time it can predict an exceeded threshold value.

When using the threshold based evaluation method there are a few important things to consider. First of all the amount of successfully predicted peaks needs to be calculated. A rather arbitrary distinction has to be made here regarding how far off the prediction can be from the actual peak for it to be considered as a successful prediction. In this case the limit for this has been set to t_f, the number of time steps in the future to predict, as is visualised in figure 3.8. The average amount of time that the model gains on reality should be calculated as well as it is a good measure of how well the model is performing. An earlier correct prediction is obviously more beneficial than a later one. However, two very important aspects to take into account is false positives and false negatives. A false positive would mean that the model predicts a peak even though it does not appear in reality. This would lead to pointlessly scaling up the process, which is an unnecessary expense. A false negative means that the model fails to predict a peak when there is one in reality. Examples of false positives, false negatives and a successful prediction is visualised in figure 3.9. The false positives should be considered more harmful than the false negatives.

In order to test different versions of the model against each other it is beneficial to have a specified comparable measurement for the performance. The three factors that affect the performance mentioned above (time, false positives and false negatives) should all be included in this score. They are not all equally important, which means that they should be weighted to reflect on that. The proposed measurement score is one where the factors are weighted by exponents. The term influenced by the false positives is called the *precision* of the model, defined as

$$P = \frac{t_p}{t_p + f_p}, \tag{3.22}$$

where t_p is the number of true positives and f_p is the number of false positives. The precision then is the percentage classified positives actually being positive. The term affected by the false negatives is very similar. It is called *recall* and is defined as

$$R = \frac{t_p}{t_p + f_n}, \tag{3.23}$$

where again t_p is the number of true positives and f_n is the amount of false negatives. The recall is the percentage of the total amount of true positives found by the model. Since $P, R \in [0, 1]$ it is suitable that the term influenced by the time factor is in the same range, so that the weighted exponents can be compared to each other in regards of the terms' importance. The time term is therefore expressed as

$$T = \frac{t_a}{t_f}, \tag{3.24}$$

where t_a is the average time gained from the prediction and t_f the desired amount of time to predict into the future (as well as the limit for a detected event to be considered a true positive). In total then we define the performance score as

$$f(t_a, fp, fn) = T^{\alpha} P^{\beta} R^{\gamma} \tag{3.25}$$

with T, P and R defined as above. In this equation the variables were set to $\alpha = 2$, $\beta = 3$ and $\gamma = 1$. This represents their importance to the results. A high exponent means lower values are punished more and in turn punish the entire score more.

The most important part is to not get many false positives (high precision), since that is connected with buying unnecessary amounts of computing power, followed by gaining a lot of time in the predictions. Getting few false negatives (high recall) is deemed the least important, since the backup solution then is just to scale up according to the current guidelines. Of course this internal ranking between the three is entirely arbitrary, as well as their relative importance, but the evaluation score gives some guidelines for comparing performance over various sets of features and differing choices of models.

4

Weibull Time-To-Event

Weibull time-to-event recurrent neural networks, or WTTE-RNN for short, was developed in Egil Martinssons' master's thesis in 2016 as a novel way to model churn prediction, but has applications under many different circumstances [3]. Unless otherwise stated, all theory in this chapter is taken from Martinsson's original thesis, in which the theory, motivations and framework for WTTE-RNN was first put together.

The key concept of the methodology is to train an RNN not to output a point prediction in every step, but to output a set of parameters θ for some probability distribution over the expected time to the next event. What constitutes an event is of course very domain specific. In the case of this thesis, an event is defined to be any point in time at which the target queue reaches the threshold on which the command to scale up the number of machines working on that specific process should be given. The TTE then is the number of minutes remaining until the next time the queue grows to that size.

One of the strengths of tuning a model to find parameters for a probability distribution instead of a point prediction is the possibility to make statements with some degree of certainty. For every time step the network generates a new set of parameters describing the probability of a new event from the current point in time and forward. Then one does not have to act based on a single guess, but can for example take action only if the possibility of an event exceeds some limit within some set time-frame.

The following sections will discuss the properties of the Weibull distribution, define the concept of censoring, give a brief reminder on likelihood functions and extend that to log-likelihood for right censored data and then describe the model used. It is restated that this will only be a brief overview of the theory, covering the essentials for understanding the methodology. For a more thorough analysis, discussion and motivation of the theory presented, the interested reader is referred to [3].

4.1 Weibull distribution

The Weibull distribution has a number of properties that are good for the proposed model. These include:

- Both continuous and discrete variants
- Unimodal
- Simple (only two parameters), but very expressive
- Numerically stable closed form CDF and Quantile function

The shape of the Weibull distribution is governed by the parameters $\alpha \in [0, \infty)$ (scale/location parameter) and $\beta \in [0, \infty)$ (shape parameter). The probability density function for the continuous case of the Weibull distribution is

$$f(t) = \begin{cases} \frac{\beta}{\alpha}(\frac{t}{\alpha})^{\beta-1} \exp[-(\frac{t}{\alpha})^{\beta}], & 0 \leq t \\ 0, & t < 0 \end{cases} \tag{4.1}$$

Two other important distributions are the cumulative hazard function

$$\Lambda(t) = \begin{cases} (\frac{t}{\alpha})^{\beta}, & 0 \leq t \\ 0, & t < 0 \end{cases} \tag{4.2}$$

and its derivative, the hazard function

$$\lambda(t) = \begin{cases} (\frac{t}{\alpha})^{\beta-1} \cdot \frac{\beta}{\alpha}, & 0 \leq t \\ 0, & t < 0 \end{cases} \tag{4.3}$$

As previously stated the Weibull distribution is very expressive, and can take many forms. When $\beta \leq 1$ the distribution is strictly decreasing. For $\beta = 1$ the distribution turns into the exponential distribution in the case of continuous distribution, or in the discrete case the geometric distribution. If $\beta \to \infty$ it converges to the Dirac delta function. Some varying cases with different α and β can be seen in figure 4.1.

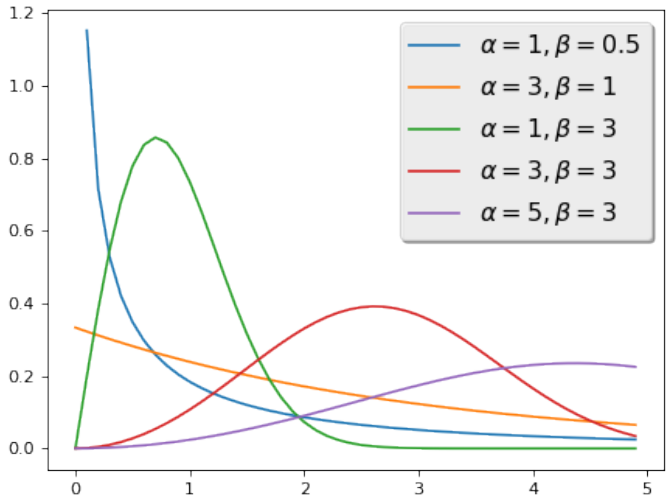


Figure 4.1: Examples of the Weibull distribution for different sets of parameters α and β.

This expressiveness means it is not necessary to make very strong assumptions on the distribution of the data examined. The Weibull distribution will probably be

able to catch some of its characteristics anyway. Perhaps it is not always a very good fit, but it should be proficient enough in finding were the large mass of probability is located and give some hints about how the probability rises and fades away over the time spectrum.

4.2 Censoring

The concept of censoring is that of not knowing when exactly an observed or examined event will happen. This is an inherent problem in waiting-time experiments. It is not certain that the examined event will occur during the time of the experiment. When looking at churn prediction this is easily imaginable, since customers might stay for a very long time, but there is certainty that they will stop being customers some time. If nothing else then by natural causes such as ageing and eventually deceasing.

Censoring then is just when an event is going to happen, or have happened, but the exact time of the event is not known. An event can be either *un-, left-, right-* or *interval* censored.

Suppose some waiting time T is had. If the exact time $T = t$ at which the event occurred is known the event is called *uncensored*. If on the other hand it is known that an event has happened before some time t, but not exactly when, so that $T \in [0, t)$ the event is called *left censored*. Similarly, if the event is known to happen after some time t so that $T \in (t, \infty)$ or in some specified time interval, $T \in [a, b]$ it is called *right censored* and *interval censored* respectively. Naturally, the types of problems studied here deal only with uncensored or right censored events.

Consider the observation (x, u) of the pair of random variables (X, Δ). Let $X = min(T, C)$ where T is the waiting time, C is the censoring variable and $\Delta = I(T \leq C)$ is the failure indicator such that

$$\Delta = \begin{cases} 0, & C < T \quad \textit{Observation is censored} \\ 1, & C \geq T \quad \textit{Observation is uncensored} \end{cases} \tag{4.4}$$

Censoring can either be *informative* or *non-informative*. The censoring is said to be informative if knowing about the censoring gives some information about the distribution of T and/or its underlying parameter(s) θ. Otherwise, the censoring is non-informative. More formally it is demanded that

$$C \perp T|\theta$$
$$C \perp \theta$$

It is assumed from here on out that only uncensored or non-informatively right censored events are dealt with.

4.3 Likelihood functions

The likelihood function is defined as the probability of some parameter(s) θ given a set of data x:

$$\mathcal{L}(\boldsymbol{x}, \theta) = f_{\boldsymbol{X}|\theta}(\boldsymbol{x}) \tag{4.5}$$

Given a set of n independent, identically distributed realisations of some measurement their joint density functions is used so that $\mathcal{L}(\boldsymbol{x}, \theta) = \prod_{k=1}^{n} f_{X|\theta}(x_k)$. Often it is convenient to instead use the *log-likelihood* function defined as

$$\log(\mathcal{L}(\boldsymbol{x}, \theta)) = \sum_{k=1}^{n} \log(f_{X|\theta}(x_k)) \tag{4.6}$$

for computational efficiency.

For non-informative right censored events the likelihood function becomes

$$\mathcal{L}(t, \theta) \propto \begin{cases} Pr(T = t|\theta) & \textit{if uncensored} \\ Pr(T > t|\theta) & \textit{if right censored} \end{cases} \tag{4.7}$$

where "proportional to" is used since additive constants are not interesting when trying to maximise the likelihood later on.

Looking at the joint PDF for (X, Δ), and omitting to write that they depend on θ for notational simplicity, leads to

$$f_{X,\Delta}(x, u) = \frac{d}{dx} F_{X,\Delta}(x, u) = \lim_{h \to 0} \frac{1}{h} Pr(\{x \leq X \leq x + h\} \cap \{\Delta = u\}) \tag{4.8}$$

Given the non-informative censoring, $C \perp T$, means that $f_{C,T}(c, t) = f_C(c) f_T(t)$. Now there are two cases to consider. First, assume $u = 0$:

$$\begin{aligned} \frac{1}{h} Pr(\{x \leq X \leq x + h\} \cap \{\Delta = 0\}) = \\ \frac{1}{h} Pr(\{x \leq min(T, C) \leq x + h\} \cap \{T < C\}) = \\ \frac{1}{h} Pr(\{x \leq T \leq x + h\} \cap \{T < C\}) = \\ \frac{1}{h} \int_x^{x+h} \int_t^{\infty} f_T(t) f_C(c) dc dt = \\ \frac{1}{h} \int_x^{x+h} f_T(t) \int_t^{\infty} f_C(c) dc dt = \\ \frac{1}{h} \int_x^{x+h} f_T(t) S_C(t) dt \to f_T(x) S_C(x) \end{aligned} \tag{4.9}$$

Thus $f_{X,\Delta}(x, 0) = f_T(x) S_C(x)$, where $S(x)$ is the *survival function* defined as

$$S(x) = Pr(x < X) = 1 - F(x) = e^{-\Lambda(x)} \tag{4.10}$$

Taking $u = 1$ and performing a similar set of calculations leads to $f_{X,\Delta}(x, 1) = f_C(x)S_T(x)$. Knowing that $C \perp T|\theta$ the joint PDF can be factored as

$$f_{X,\Delta}(x, u) = (f_T(x)S_C(x))^u \cdot (f_C(t)S_T(t))^{u-1} = [f_T(t)^u S_T(t)^{u-1}] \cdot [f_C(t)^{u-1} S_C(t)^u] \tag{4.11}$$

but it is also the case that $C \perp \theta$, meaning that $f_C(x)$ and $S_C(x)$ are both just constants with regards to θ. Thus

$$\mathcal{L}(\boldsymbol{x}, \theta) = f_{X,\Delta}(x, u) \propto f_T(t)^u S_T(t)^{1-u} \tag{4.12}$$

Utilising that $f(t) = \lambda(t)S(t)$ and $S(t) = e^{-\Lambda(t)}$ this can be rewritten to the alternate, and computationally more efficient, form

$$\begin{aligned} \mathcal{L} &= f(t)^u \cdot S(t)^{1-u} \\ &= \lambda(t)^u \cdot S(t) \\ &= \lambda(t)^u \cdot e^{-\Lambda(t)} \\ &\iff \\ \log(\mathcal{L}) &= u \cdot \log(\lambda(t)) - \Lambda(t) \end{aligned} \tag{4.13}$$

which is the final form used. The case for discrete data is developed much in the same way, eventually resulting in

$$\begin{aligned} \mathcal{L}_d &= (e^{d(t)} - 1)^u \cdot e^{-\Lambda(t+1)} \\ &\iff \\ \log(\mathcal{L}_d) &= u \cdot \log(e^{d(t)} - 1) - \Lambda(t + 1) \end{aligned} \tag{4.14}$$

where $d(t) = \Lambda(t+1) - \Lambda(t)$. Putting everything together, the discrete log-likelihood for the Weibull distribution then is

$$\log(\mathcal{L}_d) = u \cdot \log[\exp(\alpha^{-\beta}((t + 1)^\beta - t^\beta)) - 1] - \alpha^{-\beta}(t + 1)^\beta \tag{4.15}$$

4.4 The Model

As previously stated, the goal of WTTE-RNN is to, for every time-step, make a good guess on when the next event is about to happen, using historical data. Thus, an RNN is trained to output two parameters α and β. An example structure showing the procedure can be seen in figure 4.2.

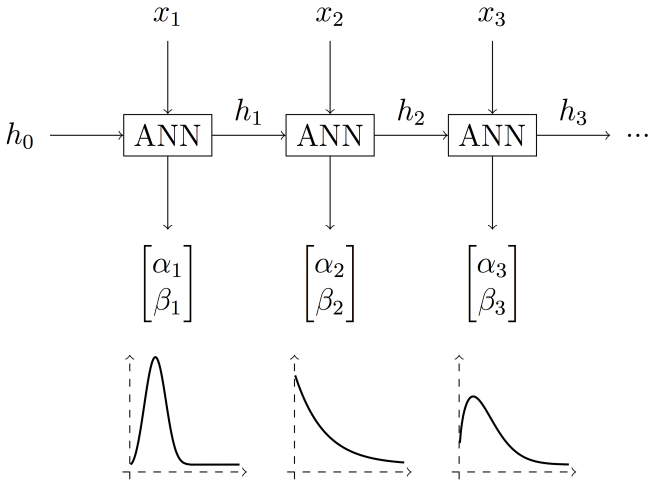


Figure 4.2: Example of structure and workings of an RNN network outputting parameters $[\alpha, \beta]^T$ for a Weibull distribution at every step. Figure taken from [1]. The x_i are new input for every time step and h_i is the internal state of the network passed on to the next time step.

A neural network is trained by minimising an energy function, often defined as the squared Euclidean distance between the output and the correct answer, using some variant of gradient based learning. Here the training is instead performed by maximising the log-likelihood for the parameters, as defined in the previous sections. Still with gradient based learning though.

Let the output of each step in the RNN be

$$\begin{bmatrix}\alpha_t\\ \beta_t\end{bmatrix} = f(x_t, h_{t-1}, w)$$

where f is a recurrent neural network, x_t the input in the current step, h_{t-1} the hidden state vector, such that $h_t = i(x_t, h_{t-1}, w_h)$ is a function of the input feature vector x_t, the previous hidden state h_{t-1} and w_h the internal weights for the hidden layer, and w the set of weights for the entire network. The optimisation problem then is to find the w that maximises the log-likelihood, adopted from equation 4.15

$$\max_w \log(\mathcal{L}_d(w, y, u, x)) := \sum_{t=0}^{T} \left(u_t \cdot \left[\exp\left[\left(\frac{y_t + 1}{\alpha_t} \right)^{\beta_t} - \left(\frac{y_t}{\alpha_t} \right)^{\beta_t} \right] - 1 \right] - \left(\frac{y_t + 1}{\alpha_t} \right)^{\beta_t} \right)$$

The optimisation is then performed via some form of gradient based learning, as explained in chapter 3.

4.4.1 Defining an event

In order to use the WTTE model it is first necessary to define what constitutes an event. In this setting, an event is considered to be any point in time at which the target queue grows to such a size that a command to add more machines to

the process would be given. Subsequent points in time, in which the queue is still larger than the assigned threshold, are not events. Instead the period immediately following the event is removed from the data. The reason for this is based on the intuition that a large queue at some point in the system is a ”failure” and subsequent data is indicative neither of a typical performance of the system under ordinary circumstances, nor the events leading up to such a point. Such data are then ”noise” or ”aftershock” from the event. Thus the complete time series is divided into several smaller time series, each ending with an event as defined above.
Then every point in time in every time series is given a Time To Event (TTE) and a label that the event is uncensored, since this thesis only works with uncensored data. This TTE, then, it what the network tries to predict. If it is more likely that an event will happen in the near future, the mass of probability close to zero will be pushed up. If it is unlikely that any event will happen soon, the mass of probability will be pushed further away.

5

Sliding Box Model

One enticing property of the Sliding Box model is its simplicity. It turns the task of event prediction into a binary classification task. For every point in time we set an *event indicator* b_t so that

$$b_t = \begin{cases} 1, & \text{if event occurred in} [t, t+\tau] \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

Based on the recent historic data the goal then is to predict whether there will be an event within the next τ time steps, for some fixed value τ.

The only modelling assumption that has to be made then is the size τ of the "box" that will be used. How far into the future would one want to look? In many circumstances, defining an optimal τ is not obvious. However, in the case of this thesis the time it takes to add more computing power to a specific process is a clear guideline on the decision. Empirically it takes 4-5 minutes from the scale up command given to the amazon web host until a rise in the number of consumers on a queue can be seen. Intuitively τ then should be set to 5 minutes.

The encoding of the data means that for every event there is a "chain" of 1's of length τ leading up to the event, and zeros otherwise. Every event then has τ points corresponding to it.

In order to classify the time points any classifying algorithm could be used. The problem could be considered as trying to estimate the parameter θ_t of a Bernoulli distribution, such that

$$B_t \sim \text{Bernoulli}(\theta_t) \tag{5.2}$$

and then, using b_t as defined in (5.1), estimate the probability

$$Pr(B_t = b_t) = \theta_t^{b_t}(1-\theta_t)^{1-b_t}. \tag{5.3}$$

Assuming that θ_t is some function of the previous data $x_{0:t}$ according to $\theta_t = g(x_{0:t})$ the task then is to maximise the likelihood of the given θ_t

$$\max_{\theta_t} \mathcal{L}(\theta_t) := \theta_t^{b_t}(1-\theta_t)^{1-b_t}. \tag{5.4}$$

The set up of the problem also means that a recurrent neural network could be trained to do the classification, meaning that the same theoretical and practical

frame work as described in the previous chapters can be used as a classifier in this setting as well, with a sigmoid activation function in the last layer of the network.

6
Results

In this chapter the performance of the three previously described models will be presented. All models have been tried on four different data sets, generated from the complete set of features using the methods described in chapter 2. These selection methods are the Naive cross-correlation, Iterative cross-correlation filtering (ICF), Granger causality and ***CLeVer*** -hybrid.

All models have been trained to detect changes on the same queue corresponding to one of the later processes in the pipeline.

The chapter concludes with presenting the effects of temporal drift on the results achieved

6.1 A quick note on selected features

No analysis will be done on precisely which features were chosen. The reasons are two-fold, but connected. Partly, the features correspond to certain metrics regarding the queues connected with certain processes. Naming the features would yield nothing to the reader, since the names give nothing but a hint on what the feature is or how it is connected to the workings of the system. On a deeper level, though, analysis could perhaps be done given that sufficient knowledge about the system was present. The authors, however, do not have such knowledge about the system in question. The amount of chosen features by each method can be found in table 6.1.

Note that ICF and Granger are completely data driven, and no intervention to the number of features chosen is done. For ***CLeVer*** -hybrid and the naive cross-correlation on the other hand, the number of features to have must be chosen manually beforehand. After running the ICF and Granger methods, and some experi-

Feature selection method	**Number of features chosen**
Naive cross-correlation	60
Iterative cross-correlation filtering	161
Granger causality	57
CLeVer	60

Table 6.1: The number of features included in each of the data sets.

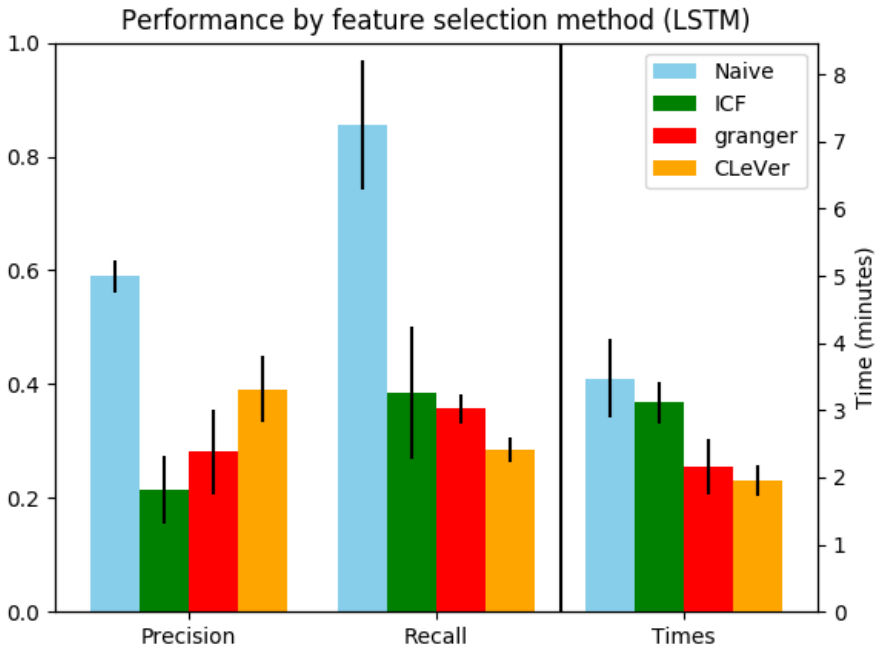


Figure 6.1: Performance for the standard LSTM model using the four different data sets. The left axis gives the scale for the precision and recall values, while the right axis measures the average time to event left when the predictions were given. The line covering the top of each bar indicates the standard deviation of the measurements.

ments of performance with different number of features, 60 was decided upon as a reasonable number. Partly because it was in line with the number of variables from the Granger causality method, and partly because it was in the upper end of how many features could be had before performance tended to drop.

6.2 Standard LSTM model

A visual representation of the results is presented in figure 6.1. A quick reminder of the concept of precision and recall perhaps ought to be given here. Precision is the percentage of all predicted positives that where actually correct. Recall on the other hand is what percentage of the actual events where captured by the model. Naturally one wants both of these scores to be as high as possible, but we value precision more than recall since false positives are more costly than false negatives.

The results show that the naive cross-correlation feature set gives a clearly better performance than the other three methods. The score, as defined in section 3.3.4, for the simulations using the different data sets can be found in table 6.2, which confirms the visual results.

Feature selection method	Evaluation score
Naive cross-correlation	0.0999
Iterative cross-correlation filtering	0.0003
Granger causality	0.0012
CLeVer	0.0017

Table 6.2: The resulting evaluation scores, as defined in section 3.3.4, for the standard LSTM model using different data sets.

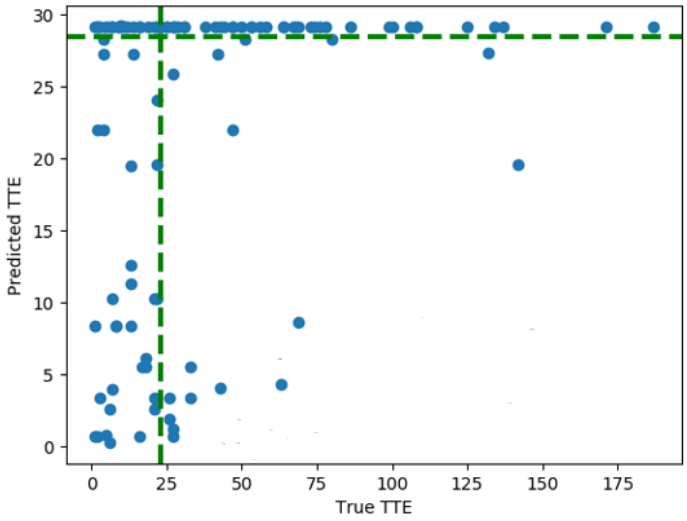


Figure 6.2: Maximum a posteriori prediction results versus true time to event from the WTTE-model. Dashed lines show the cut-off points used for deciding whether to count a prediction as false or not.

6.3 WTTE

Since the WTTE model predicts a time to event for every time-point, it has first to be decided what is a true and false prediction. First the model was trained on the training data, using a maximum look back period of 30 minutes in order to keep the data frames in a reasonable size for the computers memory limitations. For every test event then, some number of time-steps were cut off at the end, and the model was fed the test time series up until that cut off point, and then gave a guess on the distribution of true remaining time to event. The result was taken as the maximum a posteriori guess given the generated Weibull distribution. This is of course a rather arbitrary choice, and other choices are absolutely possible. The results of this for all test cases can be seen in figure 6.2.

A majority of the predictions are just under 30 minutes, the maximum look back period in the training data. However, of the predictions that are actually lower than 29 minutes, rather few are larger than 20 minutes. There is also a large spread of the points predicted to be lower than 29 in the range 0-20, and the exact results of the

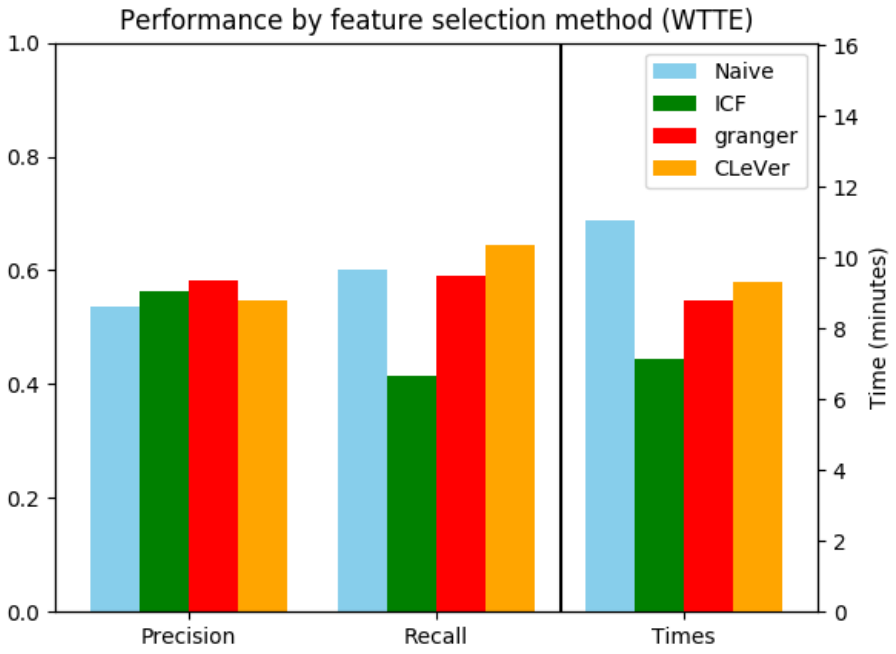


Figure 6.3: Performance for the WTTE model using the four different data sets. The left axis gives the scale for the precision and recall values, while the right axis measures the average time to event left when the predictions where given.

predictions does not seem overly trustworthy. Given the distribution of points, two cut off points (the dashed green lines in the figure) were then decided. This divides the output into four quadrants, where the lower left corresponds to true positives, the upper right to true negatives, the upper left to false negatives and the lower right to false positives. This was done in order to be able to compare the model use wise with the other tested models. This results in a possible use case where every time the model predicts an event less than 29 minutes in the future the action to scale up the number of machines could be taken, and it would be reasonably certain with some probability there would be an actual event sometime within the coming 20 minutes.

The performance results visualised can be seen in figure 6.3. The results are less clear between the data sets, but table 6.3 show that Granger causality gives the best results, followed by ***CLeVer*** .

6.4 Sliding Box

The Sliding Box model is the only one which is a classic binary classification task. Just as for the previous models, the results are similarly visualised in figure 6.4. However, the raw results are not a straight count of the number of events. The box size was set to 5, which means that for every event there are 5 points in time

Feature selection method	Evaluation score
Naive cross-correlation	0.09224
Iterative cross-correlation filtering	0.07399
Granger causality	0.11643
CLeVer	0.10508

Table 6.3: The evaluation scores describing the performance of the WTTE model when trained on data sets containing features obtained using the four different feature selection methods.

Feature selection method	Evaluation score
Naive cross-correlation	0.00692
Iterative cross-correlation filtering	0.00008
Granger causality	0.00030
CLeVer	0.00029

Table 6.4: The evaluation scores describing the performance of the sliding box model when trained on data sets containing features obtained using the four different feature selection methods.

that belong to the positive class. Thus in the extreme case, even if only one out of every five of the positive class were classified correctly, all events might have been covered. No formal check has been done on this, but a visual inspection of the results suggests the predictions from the model are often grouped together, so that if the model detects one point for an event, it often also detects some of the other points, whereas many of the events get none of their points predicted correctly.
The performance here is notably worse as compared to the other two models. Both the precision and recall are lower. Notably ICF manages to capture more of the positive class (higher recall), but only on account of making many more random guesses, managing to capture some of the true ones in the process, which is seen in the very low precision score. The total scores are also lower, as seen in table 6.4.

6.5 Temporal drift

An interesting factor when analysing time series data for predictive maintenance is how often the model needs to be retrained. Circumstances change with time and so the performance of the model will most likely change as well. Such changes of circumstances may be internal, i.e that the system itself changes by updates to specific procedures, or external, i.e. the nature of the incoming data changing. Another possible source of change is the implementation of the automatic scaling itself, which might alter the stress points of the system and redirect the surges of information to new points.

The three months of data that is available is split into the first two months for training and then the rest for testing. The test data is then consequently split into smaller intervals to determine whether the performance generally gets worse by time.

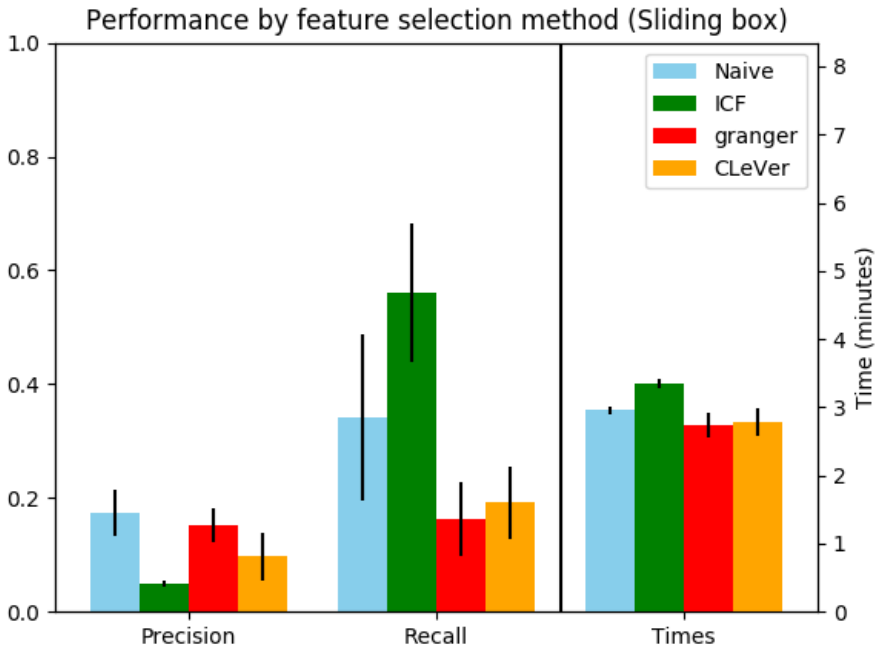


Figure 6.4: Performance for the Sliding Box model using the four different data sets. The left axis gives the scale for the precision and recall values, while the right axis measures the average time to event left when the predictions where given.

The results from these simulations are illustrated in figure 6.5. Since the simulations are rather time-consuming, they have only been performed for the standard LSTM model, but there is no reason found to suspect why the results of temporal drift would be different using another model.

There seems to be some decline in performance over the first weeks, but the rise in the score after 6 weeks of all the models gives some evidence on the contrary to this. It might be that specific types of occurrences in the system happen over time, and that some of the occurrences in the 6th week also happened a lot in the training data. More data and simulations would have to be performed in order to give some more definitive answers.

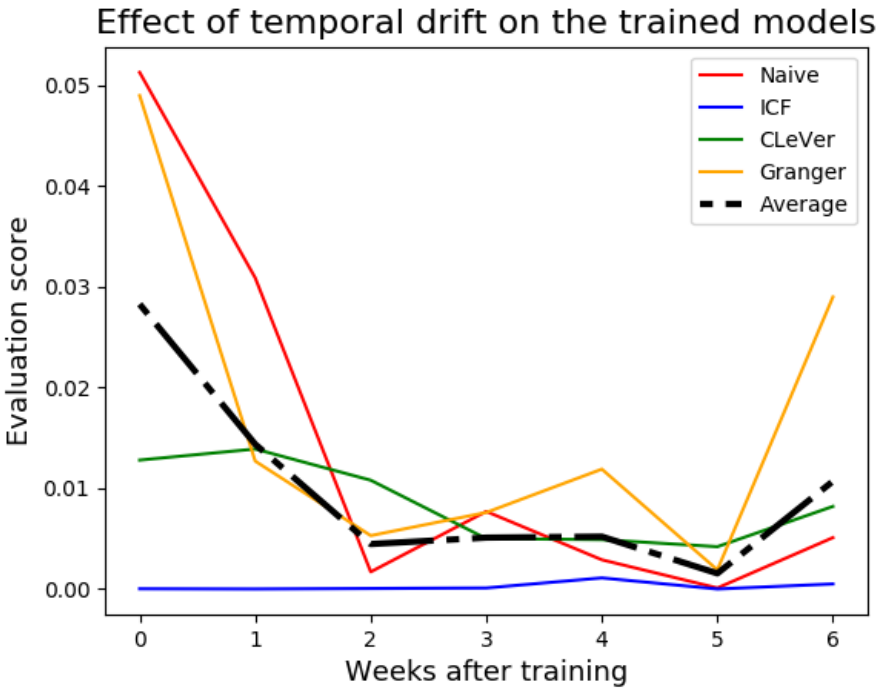


Figure 6.5: The effect of temporal drift on the standard LSTM model trained on the four different data sets. The average of all these results is also illustrated.

7
Discussion

The results in chapter 6 show that it is indeed possible to learn something about future queues building up in the system. Both the method for selecting features and the model used to look at the data does influence the results in different ways. This chapter will discuss some of the results and compare the different models against each other. It also contains some analysis on necessary actions in order to keep the models in production in spite of the drift in performance that occurs through time. The chapter is then concluded with some remarks on possible future work on the thesis.

7.1 On the importance of feature selection

What set of features are chosen from the original set obviously affect the results. Summing over all three different models, naive cross-correlation is the overall winner. For the standard LSTM model naive cross-correlation wins clearly. For WTTE on the other hand, the results are less clear. Using the evaluation scores, Granger causality comes out as the winner, closely followed by ***CL****e****V****er*. The results for the Sliding Box model were overall much poorer, but again naive cross-correlation does the best job.

Common for all three models was the fact that ICF performed worst. One suspicion as for why this is the case is that the pure number of included features does matter. The other three feature sets were much smaller (57, 60 and 60 features respectively) as compared to ICF (161 features). This was confirmed by doing simulations with using naive cross-correlation and including different amounts of features, ranging from 30 to 161. The variation in performance differed slightly in the range 30-90, and then started declining. The pure amount of information available might lead to the network not being able to find the signal and learn anything useful.

None the less, using the 161 most correlated features gave better results, as measured by the evaluation score, than using the 161 features chosen by the ICF algorithm - simulated on the standard LSTM model - which confirms that the way of choosing the features also matters.

One speculation is that the diversity of reasons that may cause queues to build up is important. An analogy could be done to a system of waterways leading from a highland to the sea. Large rainfalls on the highland causes large masses of water to

flow through the system, but it is not always known exactly which streams are going to be flooded on the way, since that varies from time to time. Perhaps something has happened in one of the streams, say a beaver has built a dam recently, which makes the water build up at that point until the dam bursts and flows further down the system. If the amount of water going to flow through a certain stream is to be predicted, the best predictor is perhaps not found by looking at the amount of water flowing through the major streams building up the entire system, but rather the streams that have historically flooded right before the one under examination. The goal then is to learn the beavers' favourite habitat so to speak.

Both naive cross-correlation and Granger causality looks at the data in just this way. By checking if the features to include tend to covariate with the target feature in some way. Naive cross-correlation follows the dam and beaver analogy even further by looking at the historical flows of information in the system, while Granger causality tries to estimate the features predictive power as well. On the other hand ***CLeVer***, and ICF, try to build a set of features that span the feature space as good as possible.

It is often the case that one tries to find such a set that most efficiently spans the possible features space. One suspicion as to why that is not ideal in this scenario is the sheer number of reasons why faults may occur and the complexity of the system. It is perhaps difficult to find a set of features that describe the entire system efficiently and capture enough information to make predictions on for a specific process. Thus maximising the amount of information available for that process by taking the highest cross-correlating features for that process is better, despite possible problems with features "stealing importance" from each other, as is often something one wants to avoid. Without looking at the right streams of information, nothing can be learned.

7.2 Evaluation and comparison of the models' performances

Results in the previous chapter confirm that how one chooses to look at and evaluate the results greatly impacts what the RNN can learn from the very same information. One initial hypothesis was that the simplicity of the evaluation would affect results. However, this proved not to be the case. The Sliding Box model, which is arguably the most simplistic, reducing the problem to distinguishing between two classes, performs worst with a large margin. Predicting queue sizes in a range from 0 to theoretically infinity using the standard LSTM model on the other hand does a much better job, and the most complicated model, trying to generate parameters for a probability distribution over an arbitrarily defined time to event, gives the highest evaluation score.

One guess is that the assumptions that a given time series of some length would belong to one of two classes is too simplistic and does not at all capture the complexity

required by the problem. Pointing at what exactly a neural network really learns is difficult task. One reasonable guess then could be that the RNN picks up on trends and changes in the queue sizes and their rate of flow of messages. Thus using an evaluation metric (queue size of a specific queue) similar to the input metrics (queue sizes and their rate of change for some other sets of queues including the target one) would perform better than abstracting the output to something perhaps simpler, but conceptually different from what was learned on.

Although the WTTE model actually achieves the highest score of all on one data set, this does not necessarily mean it is the best one to use in production. There are some caveats to discuss. The first and foremost being how to decide to take action on a prediction. To reiterate from the previous chapter, the way this was done in the testing was by taking the maximum a posteriori prediction based on the generated probability distribution. Then some thresholds were decided based on the distribution of the test predictions. This threshold must be set rather high in order to capture a reasonable number of events. In our case, the threshold was set to 29 minutes for the prediction and 20 minutes for the true results. In production this would mean that every time an event was predicted to be closer in time than 29 minutes, an action would be taken to scale up the amount of computers working on that queue. With some probability then, there would actually be an event within the next 20 minutes. On average this would occur sometime between 7-11 minutes after the prediction was made. This means that the scaling of the computing power would on average be done 2-6 minutes ahead of when necessary, and for the false predictions it would be kept for a total of 20 minutes before being shut down.

What kinds of events get missed and what kinds of events get predicted correctly has not been studied. It could be that all models miss the same events, or they could have differences. Perhaps some clustering of the events based some proximity measure could be done. Further studies needs to be done in order to confirm or discard this.

It must also be noticed that the way the evaluation score was designed and weighted favoured the WTTE results over the LSTM results, in that T is set to 1 for all WTTE models given the very early prediction done. Predicting more than 5 minutes ahead of time is not an advantage, but rather a disadvantage. Designing the score in another way, for example using an additive rather than a multiplicative model, and adjusting the weights would change many of the rankings between different sets of features and models. The exact trade-offs one would be willing to make and what matters most would have to be considered before choosing an optimal model and feature set to do further work on.

7.3 Temporal drift

As time goes on the circumstances surrounding the system change. New things are being deployed every week, which implies that the system is constantly changing and will behave different than before. In effect, this means that the data the model has

been trained on will be less and less representative of the system as time passes. To deal with this problem the model needs to be retrained on more recent data, which in turn induces some complications. The models are trained on different metrics surrounding the queues in the system. These queues will be directly affected when the model is being used. The model will inform the system to signal for scaling up earlier than it usually does. This will lead to the corresponding process having the computing power that it needs at an earlier stage and therefore preventing the queue to become as large as it otherwise would have. The new and different characteristics of the queues will be unknown to the model which has been trained on data in a dissimilar situation. This in itself affects the temporal drift negatively. The severity of the effect is difficult to accurately, or at all, measure before the models are actually in production.

This opens up a possible discussion about how often the models would need to be retrained. The problem with retraining the models is that they would be retrained on data that has been influenced by the currently used model. To emulate the same conditions as during the training, the newly trained model would have to be used on top of the previous model. Stacking models this way is obviously not sustainable, especially if retraining is necessary after a short amount of time. From the results in figure 6.5 it is difficult to say how often retraining would be necessary. After just a couple of weeks the average performance of the models has decreased significantly. However, the rise in performance after 6 weeks suggest that perhaps similar events that occurred in the training data did not occur in weeks 2-5, but did in week 1 and 6. To determine what is really temporal drift and what is just chance behaviour would need more extensive testing.

Based on figure 6.5 one could suggest that the models would need to be retrained at least once every two weeks. This would quite quickly lead to a lot of models being stacked on top of each other. A rather naive solution to this would be to let the models stack until it could cause problems and then start from scratch, meaning that new data when the model is not being used is collected and trained on. Quite a tedious process considering that we started with three months' data. However, by this logic the system has changed too much during two weeks after the training stopped for the model to be useful. Is it by the same logic viable to use three months' data for training? The behaviour of the system could have changed so much from the beginning of that period to the end that there might be a lot of the data that can be considered as worthless. The problem is that with data from a shorter amount of time there are not enough significant events to train on. Many aspects can influence the queue metrics, which implies that the models need to train on all of these different situations in order to be able to make a sufficient prediction when a similar situation appears.

7.4 Concluding remarks

From the work in this thesis it is evident that learning when to scale up computing power ahead of when it is actually needed is indeed possible. Selecting features based

on their connection to the target feature seems to give better results than selecting features that best span the entire feature space. One reason for this might be the complexity of the system. Using the standard LSTM model with features chosen with naive cross-correlation or the WTTE model with features from Granger causality gives the best results, but the models have different strengths and weaknesses that would have to be considered. Some evidence of temporal drift in the results are found, but again, more extensive testing has to be done in order to confirm or discard this.

To consider putting the models into production, a choice first has to be made on what importance the precision, recall and time gained in the prediction have. Then more model tuning should be done, optimising various hyper-parameters connected to the model.

As in all machine learning task, collecting more data would of course also be beneficial. It should also be tested whether 1 minute sampling of the data is indeed the best, and if so how long the time series fed to the network should be. Perhaps higher frequency data can give even more information, or perhaps that is redundant information. Memory management in computers have to some extent been a limiting factor during the work with this thesis, but more extensive testing could be done given more resources.

Bibliography

[1] E. Martinsson, "WTTE-RNN - Less hacky churn prediction," https://ragulpr.github.io/2016/12/22/WTTE-RNN-Hackless-churn-modeling/, accessed: 2018-04-16.

[2] O. Ibidunmoye and E. Elmroth, "Blackbox strategies for detecting service performance anomalies in virtualized environments," 2016.

[3] E. Martinsson, "WTTE-RNN : Weibull Time To Event Recurrent Neural Network," Master's thesis, Chalmers University Of Technology, 2017.

[4] R. Bellman, *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[5] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1, pp. 273 – 324, 1997, relevance. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000437029700043X

[6] B. Davide, "Unsupervised feature selection for sensor time-series in pervasive computing applications," *Neural Computing and Applications*, vol. 27, no. 5, pp. 1077–1091, 2016. [Online]. Available: http://pages.di.unipi.it/bacciu/wp-content/uploads/sites/12/2016/04/nca2015.pdf

[7] C. W. J. Granger, "Investigating causal relations by econometric models and cross-spectral methods," *Econometrica*, vol. 37, no. 3, pp. 424–438, 1969.

[8] Y. Sun, J. Li, J. Liu, C. Chow, B. Sun, and R. Wang, "Using causal discovery for feature selection in multivariate numerical time series," *Machine Learning*, vol. 101, no. 1, pp. 377–395, Oct 2015. [Online]. Available: https://doi.org/10.1007/s10994-014-5460-1

[9] H. Yoon, K. Yang, and C. Shahabi, "Feature subset selection and feature ranking for multivariate time series," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 9, pp. 1186–1198, Sept 2005.

[10] W. J. Krzanowski, "Between-groups comparison of principal components," *Journal of the American Statistical Association*, vol. 74, no. 367, pp. 703–707, 1979. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1979.10481674

[11] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: a tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, Mar 1996.

[12] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computationr*. CRC Press, 1991.

[13] B. Kröse and P. van der Smagt, "An introduction to neural networks," 1993.

[14] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[15] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[16] D. Wackerly, W. Mendenhall, and R. Scheaffer, *Mathematical statistics with applications*. Nelson Education, 2007.

[17] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization." *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1412.html#KingmaB14

[18] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1999.

[19] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International Conference on Machine Learning*, 2013, pp. 1310–1318.

[20] A. Gers F., J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," Tech. Rep., 1999.

[21] Z. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," 05 2015.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735