Predictive Maintenance Using Machine Learning Methods in Petrochemical Refineries

Sander Suursalu

CE-MS-2017-07

Delft University of Technology

Abstract

This research project evaluates the suitability of machine learning methods for early fault prediction and predictive maintenance in petrochemical refineries based on real-life use cases at Shell Pernis. Refineries are mature industrial installations, however, unplanned shutdowns still occur due to equipment failures. Refineries have petabytes of process control data available from the past years, however, all of that data is unlabelled. The goal of this research project was to evaluate, whether useful information can be extracted from the process control data. The resulting approach had to be compatible with Shell IT, scalable to larger sections of the refinery, reusable in other parts of the refinery and capable of detecting the components that cause the potential faults. During this research project, multiple solutions based on artificial neural networks and statistical approaches were implemented to model the normal behaviour of the monitored systems. Abnormal predictions for the modelled systems were then used to predict failures in advance, where the prediction horizon reached more than a month for some use cases. 4-layer GRUs with *tanh* activation functions and an input sequence length of 4 samples provided the best results. GRUs were 7% faster to train than LSTMs while reducing the prediction error by 15%. Furthermore, the prediction error was less than 3% for the normal operating conditions while reaching more than 15% prior to failures. Therefore, machine learning models can predict failures in petrochemical refineries without any industry-specific knowledge, if the model is trained with clean data that does not contain abnormal time series.

TU Delft Delft University of Technology

CE Lab Computer Engineering Laboratory

Predictive Maintenance

Using Machine Learning Methods in Petrochemical Refineries

by

Sander Suursalu

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday July 11, 2017 at 03:30 PM.

Student number:	4512693	
Project duration:	November 1, 2016 – July 11, 2017	
Thesis committee:	Dr. Z. Al-Ars,	TU Delft, supervisor
	Dr. C. Hauff,	TU Delft
	Ing. P. Kwaspen,	Royal Dutch Shell

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

This research project is the result of a collaboration between Delft University of Technology and Shell. It was made possible by the availability of an internship at Shell that provided access to the required infrastructure as well as the datasets. Therefore, I would like to thank Shell for this possibility as well as my supervisor ing. P. Kwaspen for his support. I would like to thank my academic supervisor dr. Z. Al-Ars for proposing this research project as well as for his guidance during this research.

Additionally, I would like to thank Skype Technologies, Cybernetica and ESF USA for the scholarships that I was awarded with during my studies at Delft University of Technology.

Finally, I would like to thank my parents for motivating and guiding me throughout the years.

Sander Suursalu
Delft, July 2017

Contents

1 Introduction **1**

2 Case Study **5**
2.1 Crude Distiller . . . 5
2.2 Control Valve . . . 7
2.2.1 Use Case 1: Slowly Loosening Valve . . . 9
2.2.2 Use Case 2: Manual Bypass Valve . . . 9

3 Machine Learning **11**
3.1 Basics . . . 11
3.2 Fault Detection . . . 12
3.3 Linear Regression . . . 13
3.3.1 Ordinary Least Squares . . . 13
3.3.2 Ridge Regression . . . 14
3.3.3 Lasso Regression . . . 14
3.4 Support Vector Machines . . . 15
3.5 Artificial Neural Networks . . . 16
3.5.1 Deep Feedforward Neural Networks . . . 18
3.5.2 Autoencoders . . . 18
3.5.3 Recurrent Neural Networks . . . 20
3.5.4 Convolutional Neural Networks . . . 22
3.6 Summary . . . 24

4 Scalability **27**
4.1 Specialised Hardware . . . 27
4.1.1 GPUs . . . 27
4.1.2 ASICs . . . 27
4.2 Parallelisation Strategy . . . 29
4.2.1 Model Parallelism . . . 29
4.2.2 Data Parallelism . . . 30
4.2.3 Distributed Training . . . 31

5 Implementation **33**
5.1 Technology Stack . . . 33
5.1.1 Amazon Web Services EC2 . . . 34
5.1.2 Databricks . . . 34
5.1.3 Apache Spark . . . 35
5.1.4 TensorFlow . . . 36
5.2 Machine Learning . . . 37
5.2.1 Pipeline . . . 37
5.2.2 Recurrent Neural Networks . . . 38
5.2.3 Deep Feedforward Neural Networks . . . 40
5.2.4 Ridge Regression . . . 41
5.3 Scalability . . . 41
5.3.1 Model Parallelism . . . 41
5.3.2 Data Parallelism . . . 42

6 Experimental Results **45**
6.1 Design Space Exploration 45
6.2 Use Case 1 47
6.3 Use Case 2 50
6.4 Normal Data Selection 51
6.5 Scalability 52

7 Conclusions & Future Work **55**

A Naming Conventions **57**

Bibliography **59**

1 Introduction

New technologies developed by the oil and gas industry are largely driven by oil prices. Average oil prices were on the rise during the last decades, which led into the development of extremely expensive technologies for oil production, e.g. the Floating LNG platform developed by Shell. However, the oil price plummeted in mid-2014 and has stayed relatively low since then. Reduced oil prices do not justify expensive projects for the development of new extraction techniques, instead, oil companies are striving for the increased efficiency of already existing utilities.

The oil and gas industry is divided into three sectors: upstream, midstream and downstream. The upstream sector performs the exploration of new crude oil and natural gas fields both underground and underwater. However, large crude oil reservoirs that have been found are often left untapped for decades, as the production cost would have been too high for the respective oil prices. Therefore, a production site will only be constructed if a new field has been found, that makes the production financially feasible. Then, a variety of hydrocarbons can be produced for years until the majority of the resources has been extracted. The refining of crude oil as well as the processing of natural gas is performed in downstream plants, which make the raw produce from the upstream sector into useful and marketable products. The midstream sector, as its name suggests, connects the upstream extraction sites with the downstream manufacturing sites, which includes the storage as well as the transportation of the hydrocarbons using pipelines, trucks, railroads and oil tankers.

Refineries are mature installations, that have not changed significantly in the last decades while still producing a commodity product. Therefore, refineries have to be operated extremely efficiently in order to increase their earnings margin while staying competitive. Despite being a mature industry, refineries are still frequently shut down due to unscheduled events. Slaughter, et al. (2015) [8] have estimated the cost of unscheduled refinery shutdowns worldwide for $20B every year. Furthermore, inefficient maintenance practices increase the operational costs by $60B every year.

Typically, maintenance is scheduled for fixed periods, i.e. turnarounds (TAs), where the entire refinery is overhauled. During TAs, refinery workers inspect the components and perform repairs if required. TAs might be scheduled for a few times per decade, as they require weeks or even months of work during which the refinery has to be shut down. This approach is known as preventive maintenance, as components are evaluated or replaced based on predefined intervals. It is presumed, that all components can survive the entire period between TAs, however, it is impossible to guarantee such behaviour for a refinery that has thousands of components. Therefore, refineries are keen on looking at technologies that could provide them with early fault notifications. Ideally, the predictions should be complemented with the time horizon for which the component is still capable of functioning, as this would enable the operators to evaluate whether the degrading component can perform reasonably until the next TA. Recent trends in this direction have been collectively named predictive maintenance, as they predict the maintenance requirements based on the actual conditions rather that a set of predefined rules.

A common overview of process analytics categories is presented in Figure 1.1. Descriptive methods

are the simplest approach to process analytics, as they provide information about what has happened. It might include reports about prior events as well as dashboards for operational monitoring. Diagnostic methods take this approach further by providing the operators with information about why something happened. Common diagnostic approaches provide operators with tools that enable them to visualize the data or search for patterns in the historical datasets. Predictive methods provide insights into what will happen in the future instead of focusing at events that already happened. However, they do not state the correct sequence of actions that should be carried out when an anomaly is detected. This is achieved by prescriptive methods, that learn the industry-specific knowledge that is commonly known only by specialists. The shift from descriptive methods towards prescriptive methods not only increases the efficiency of the refineries, but paves the way for reduced manning in the operations. Eventually, this could lead to normally unmanned installations in a variety of petrochemical facilities.

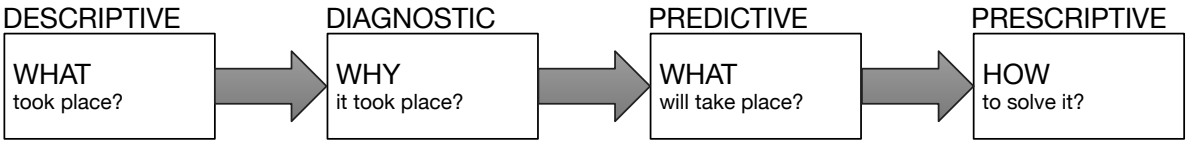


Figure 1.1: Methods for process analytics

Fault detection methods are commonly categorized into data-driven and model-based approaches. Data-driven methods use historical process control data to detect new samples that are not in the normal operating boundaries of the process. It does not require any knowledge of the underlying processes, however, the selection of the normal datasets is of utmost importance. A common data-driven method is limit-checking, where the upper control limit (UCL) and lower control limit (LCL) are devised based on the normal operating conditions of a process, as illustrated in Figure 1.2. A fault can be detected whenever the process values cross the upper or lower bounds. Multiple process variables can be dependent on each other, which means that the behaviour of one variable significantly affects the other. Multivariate limit-checking can be used to improve the performance of the fault detection method under such circumstances. Although data-driven methods are widely used in refineries, their simple construction prohibits them from providing the operators with more complex fault prediction methods.

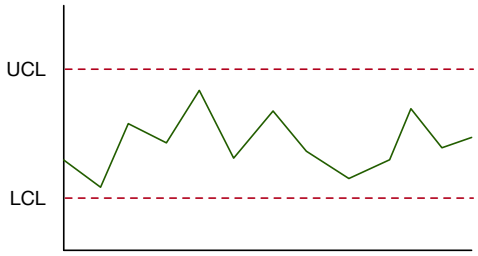


Figure 1.2: Data-driven fault detection

Model-based approaches devise a model of the underlying process, which can then be used to detect faults. A common approach for fault detection using physical models is presented in Figure 1.3. The inputs and the outputs of the process are used to create a physical process model. Then, features that can be compared to the normal behaviour of the process are generated. Finally, an error can be detected based on a fault detection mechanism. Model-based methods can be created using a variety of approaches. Physics can be used to create theoretical models that represent the underlying physics of the monitored process. Theoretical approaches can be used to model all of the possible operating conditions of a system, however, the construction of such models is often too complex and costly for large systems. Therefore, machine learning approaches have been recently used to create models based on experimental data. These methods evaluate the relations between the inputs and outputs to devise an approximation of the underlying system. The quality and quantity of the available data has a significant effect on the obtained model, as these methods use real-life datasets to create a model of the system.

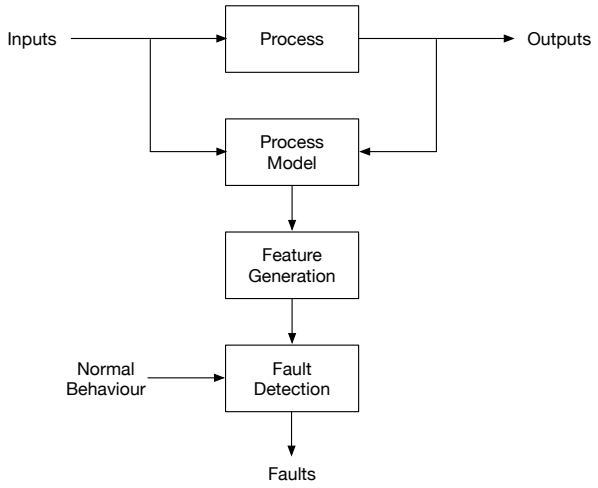


Figure 1.3: Model-based fault detection

A model-based fault prediction method was created during this research project, which was based on an experimental dataset. The shift towards more complex prediction methods is largely driven by the advancements in machine learning and large-scale computing during recent years. This research project evaluates the applicability of novel deep learning approaches for petrochemical installations based on a real-world use case from a downstream operation, namely the Shell Pernis refinery in Rotterdam, the Netherlands. Shell Pernis is the largest refinery in Europe [17], which uses more than 50 plants to process 20M tonnes of crude oil every year. More than 50,000 sensors have been installed in the refinery that are used for process control. The sensor measurements are stored in a data historian that has more than 10 year's worth of time series measurements available, resulting in roughly a petabyte of data. The measurements are stored in the data historian only once per minute, although the sensors are sampled at a higher rate. This discards a large amount of data, because the stored values only represent the behaviour of the process for a fraction of the time between two samples. It has to be noted, that the dataset is based on numerical process control data, therefore, it does not include any information about the status of the components or failures.

The objective of this research was to evaluate whether a generic machine learning approach can be constructed that is capable of predicting failures in petrochemical refineries. The approach had to produce early predictions without any knowledge of the underlying processes in order to be reusable for arbitrary petrochemical processes. Therefore, the approach had to be able to automatically extract useful information from a large collection of input features that might affect any given output variable. The input feature set was a collection of numerical values from the data historian database. As the measurements are performed once per minute, the process control data is not capable of representing fast transitions in the petrochemical processes, such as the behaviour of closed-loop controllers. Furthermore, the machine learning approach had to be capable of handling highly correlating process variables, which might be present if multiple processes are duplicated in parallel. The long-term goal of the project is to apply early fault detection and predictive maintenance methods for the entire refinery, therefore, the approach had to perform well for thousands of input features. The number of input features is dependent on the battery limits, which define the first and the last component of the modelled section of the refinery via which the hydrocarbons flow. This means, that the machine learning model had to make predictions for multiple process variables within the battery limits in order to detect the specific section of the monitored processes that behaves erroneously.

Three main research questions can be formulated based on the objectives of the project:

1. *Is it possible to predict failures in petrochemical refineries using machine learning approaches based on the process control data for the normal operating conditions?*

2. *Can failures be predicted at least a week in advance?*

3. *Do machine learning methods that use time series data as inputs outperform methods that use discrete values as inputs?*

The report is structured into 7 chapters. Chapter 2 provides the background on the refinery and the use cases. The machine learning approaches that could be applied for the use case are discussed in Chapter 3. Chapter 4 provides an overview of the approaches that could be taken in order to make the training phase scalable. Chapter 5 provides an overview of the technology stack as well as the details about the implementation. Finally, the experimental results will be provided in Chapter 6 with the corresponding conclusions and recommendations about the results in Chapter 7.

2

Case Study

This chapter presents a description of the case study as well as an overview of the Shell Pernis refinery. Section 2.1 discusses the high-level architecture of the modelled system, which places the use cases into context. Section 2.2 provides an overview of control valves as well as the common failure modes and use cases.

2.1. Crude Distiller

Petrochemical refineries are complex installations, the core of which is a sequence of distillation units that refine crude oil into useful products. Crude oil consists of a variety of hydrocarbons that have different boiling ranges. Fractional distillation units utilize these differences in order to separate the incoming crude into multiple outgoing hydrocarbon flows. At Shell Pernis, the refining process is started by passing crude oil through two crude distillers, which separate it into a range of hydrocarbons: fuel gas, C3 & C4, tops, naphtha, jet fuel, LGO & NGO.

These hydrocarbons are further processed by a number of processing units, which together form a branched flow of hydrocarbons. All of the crude is passed through these crude distillers as they are at the root of the branching structure. Therefore, these crude distillers are one of the most critical components in the refinery, since they affect all of the other processes.

This research project will be performed based on the use cases from one of these crude distillers, also known as crude distiller XYZ (referenced as XYZ in the thesis due to confidentiality). It is a complex process unit, which has more than 7,500 process variables available. Therefore, it is further structured between multiple larger units, as illustrated in Figure 2.1. The refining process is started by pumping crude oil from the storage tanks to the desalter, which is used to remove the salt from the water that is dissolved in the crude oil. Then, the crude is heated in the furnace in order to increase its temperature to the required value set by the primary distillation column. The primary distillation column will split the crude into multiple intermediate products, that will be further processed in the XYZ unit. Only the construction of the furnace will be further described, because it was the point of failure for the use cases.

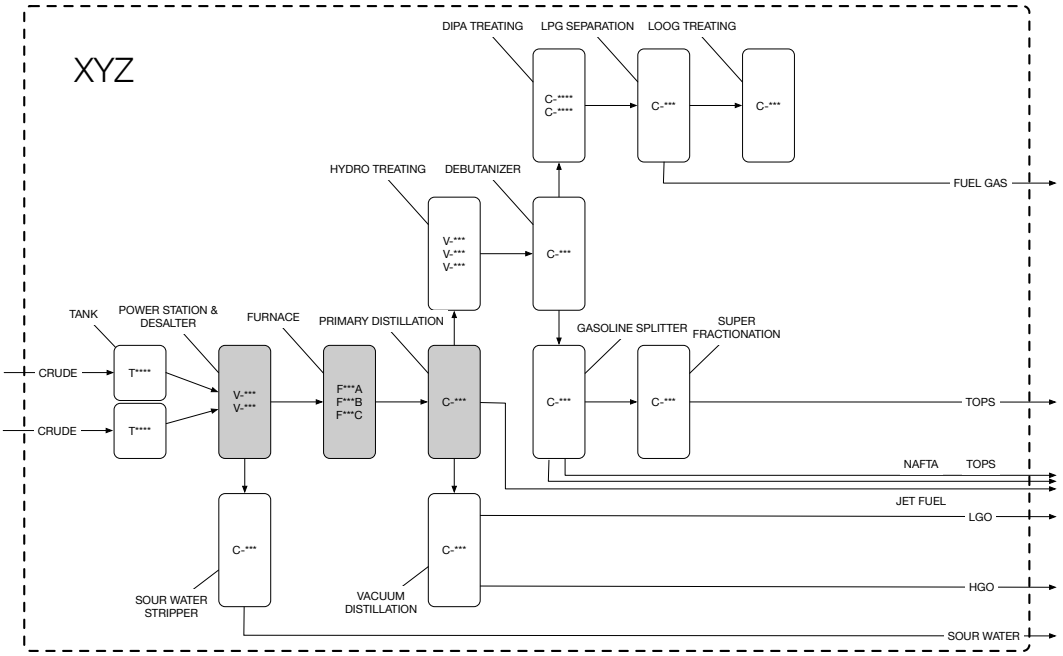


Figure 2.1: Schematic overview of the processes in crude distiller XYZ, where the greyed units are modelled in this research project

Ideally, furnaces could be simple installations that increase the temperature of a fluid that is passed through its radiant section. In practice, the feed stream has to be split into multiple passes to improve heat transfer and to achieve uniform heat distribution within the crude. At Shell Pernis, the furnaces have * passes, although 2 pass furnaces are common elsewhere [9]. The streams in the passes are recombined into a single stream after the furnace, as illustrated in Figure 2.2. Furthermore, the furnace in the XYZ unit has been divided into three sections: A, B, and C with *, *, and * passes, respectively. The global outlet temperature is set by the requirements of the downstream process, e.g. a distillation column. The temperature is regulated by a control valve that adjusts the amount of fuel gas supplied to the furnace. Additionally, the amount of crude passed through a pass to a section of the furnace is controlled, because this enables the furnace to fine-tune the temperatures at each pass. The flow at each pass, that is measured in tonnes per day (t/d), is changed using control valves in response to the measured changes in the temperature of the crude and based on the temperature differences between parallel passes. This is required to achieve a unique heat distribution between the passes, since the structural and behavioural differences (e.g. coking, which is the buildup of coke due to too high temperatures) of passes can otherwise cause deviations between passes.

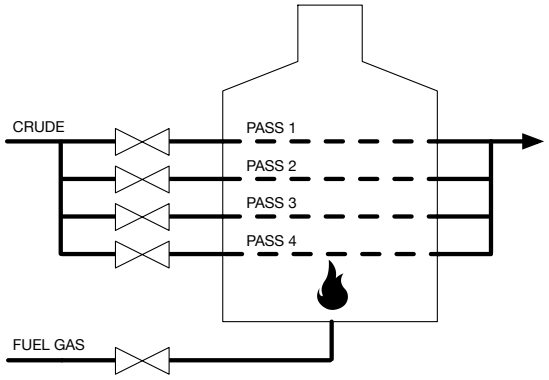


Figure 2.2: A furnace with multiple passes for efficient heat transfer

One of the goals of this research project was to devise an anomaly detection method that is capable of learning from a large set of features without any specific inputs from industry experts. Therefore, all

of the numeric parameters from the furnace that are stored in the historian database will be used as inputs for the machine learning algorithm. Furthermore, a small number of parameters that are located directly before or after the furnace will be included in the inputs to the model, i.e. the feature set. This results in 184 features out of which 148 features are real-valued and 36 from a discrete set of values as listed in Table 2.1. It is sufficiently large to prove whether machine learning can be used for early fault detection in petrochemical refineries, yet small enough to implement a solution without a too large overhead from long training times. The scope should be extended to see whether the same approaches can be used for the entire crude distiller unit, if the current solution fulfils the requirements at Shell Pernis.

Parameter names refer to its type as well as the unit to which the measurement belongs. It can be generalized to the following form: [unit]:[process].[type], where unit is the high-level structure, e.g. XYZ; process refers to specific physical devices, e.g. valves, gauges, controllers; and type refers to one of the available parameters for the specified process. Common process names include FC, TC, PC for flow, temperature and pressure controller, respectively; FI and TI refer to flow and temperature indicator; FT and TT refer to flow and temperature transmitter.

Parameter Name	Quantity	Unit	Description
XYZ:***FC***.SP	20	t/d	Flow, setpoint
XYZ:***FC***.PV	20	t/d	Flow, measured
XYZ:***FC***.OP	20	%	Valve position, direct
XYZ:***FC***.MODE	20	{Man, Cas, Aut}	Flow controller mode
XYZ:***FI***.PV	16	t/d	Flow, measured
XYZ:***TC***.SP	16	°C	Temperature, setpoint
XYZ:***TC***.PV	16	°C	Temperature, measured
XYZ:***TC***.OP	16	%	Valve position, cascaded
XYZ:***TC***.MODE	16	{Man, Cas, Aut}	Temperature controller mode
XYZ:***TT***.PV	20	°C	Temperature, measured
XYZ:***TI***.PV	1	°C	Temperature, measured
XYZ:***PT***.PV	1	Bar	Pressure, measured
XYZ:***LT***.PV	1	%	Level, measured
XYZ:***LZ***.PV	1	%	Level, safeguard
Total:	184		

Table 2.1: Process variables used for machine learning

2.2. Control Valve

The physical devices that caused the analysed issues in the furnace are control valves, therefore, their construction will be presented in more depth. Control valves are essentially closed-loop controllers that modify the position of the valve opening and thereby control the fluid flow. This makes it possible to automatically regulate the flow in response to measurements of process parameters in an industrial plant. Closed-loop controllers modify the opening of the valve (OP) in order to keep the process variable (PV) at the desired setpoint (SP). Therefore, they are capable of keeping the flow at the desired value even under large disturbances and changes in the system. However, closed-loop controllers are not capable of compensating for the disturbances under all circumstances. These situations cause faults at the refinery, which might force the operators to shut down the plant.

A variety of valve types are used in petrochemical refineries, e.g. sliding-stem valves and rotary valves [9]. Control valves are composed of multiple components: a valve, that regulates the flow; an actuator, that alters the valve's position; a positioner, that powers the actuator based on the signal from the controller; and a controller, that calculates the required positioning.

An important property of valves is its flow characteristic, which describes the relationship between the flow through a valve and its stem opening. Most of the control valves are either linear, where the flow increases linearly with the stem opening; equal percentage, where the flow increases exponentially with the stem opening; or modified parabolic, which is a compromise between linear and equal percentage control valves [19]. Equal percentage control valves are used in the analysed units, therefore, the control valves are non-linear in their global operating range. However, equal percentage control valves can behave linearly in local operating ranges, if the valve opening fluctuates only within a small range.

A typical control valve schematic is presented in Figure 2.3, where electrical signals are represented by dashed lines and crude pipes by solid lines. Control valves are controlled by a flow controller, which regulates the flow through a pipe in order to minimize the difference between the target flow and the measured flow. The sensors that measure the flow are typically separate from the flow controller. Similarly, the flow controller does not regulate the valve directly, but transmits a signal to the valve positioner.

Bypass valves are manual devices that are frequently installed in parallel to control valves, in order to perform maintenance on the control valves without shutting down the refinery. Bypass valves are normally closed and have to be opened manually by refinery workers. However, bypass valves are not usually utilized for critical valves, e.g. the control valve regulating the fuel gas supply in Figure 2.2, due to potential safety hazards. Early anomaly detection is therefore highly valued for critical valves, since they are not usually repaired while the processes are continued. It has to be noted, that the position of bypass valves is not stored in the historian database. This can create unusual patterns in other logged process variables, because the refinery will still be operational although the processes do not match their standard operating patterns, as illustrated by c) in Figure 2.4.

Control valves can cause a number of potential issues in refineries, some of which are illustrated in Figure 2.4. The use cases evaluated in this research project are based on examples b) and c). It is important to notice, that the closed-loop controllers can compensate for the disturbances and changes in the system for months. Therefore, the behaviour of control valves can be abnormal even when the refinery is operational. The solutions presented in this research project are searching for these early notification signs, which can then be used to predict future failures. However, it makes the training phase of machine learning significantly harder, as the control valves might not behave normally for months although they do not cause a fault in the refinery.

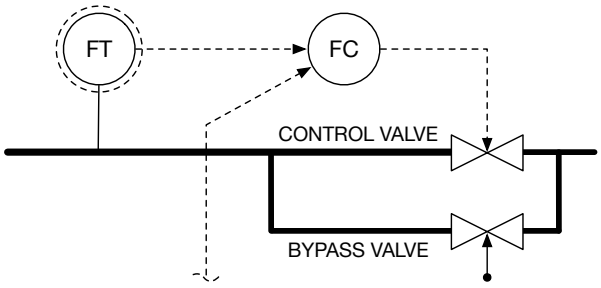


Figure 2.3: Typical control valve setup in a refinery, where the flow transmitter (FT) measures the flow and the flow controller (FC) regulates the flow by modifying the opening of the control valve based on the setpoint and the process variable

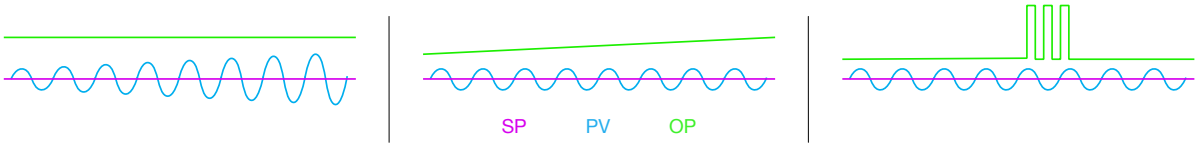


Figure 2.4: a) The process variable oscillates with an ever-increasing amplitude b) The controller has to increase the opening of the valve to keep the flow at the specified setpoint c) The controller output saturates due to a bypass valve being open

2.2.1. Use Case 1: Slowly Loosening Valve

The first anomaly that is being investigated in this research was caused by the slow detachment of the plug and stem connection of a control valve over a period of 90 days. The issue was detected by the refinery workers, who noticed an oil leakage close to a heavily vibrating valve. The vibrations were caused by the detached connection between the plug and stem, which allowed for parts of the valve to move excessively. Abnormal vibrations cracked a welding joint slightly after the control valve. This anomaly caused a shutdown of half of the refinery for several days, therefore, it was thoroughly analysed by industry experts to see, whether it could have been predicted in advance.

It was observed that the ratio between the setpoints of all the flow controllers in the F***B furnace were stable prior to the incident, as illustrated in Figure 2.5. It is clearly visible that the first and the second as well as the third and fourth control valve are highly correlated. However, it was noticed that the flow controller for the control valve that caused the issue had to open the valve increasingly more over a duration of 90 days in order to keep the flow at the desired level. Therefore, it would have been possible to detect the issue months in advance, as can be seen from Figure 2.6. However, it is impossible to manually monitor such transitions, therefore, machine learning approaches were evaluated in this research project to see, whether it could be performed in an automated way.

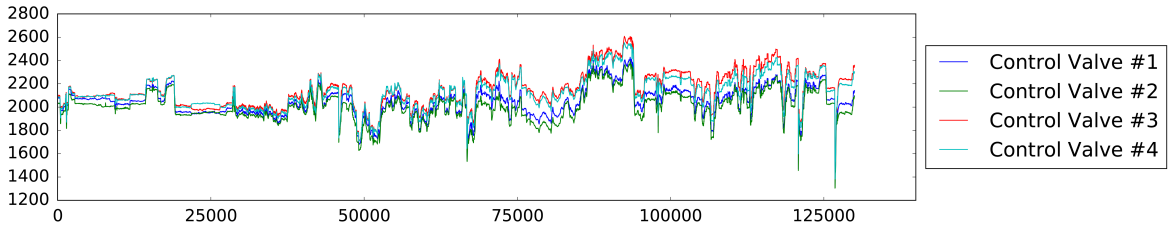


Figure 2.5: Setpoints for all FCs in F***B have the same relationship over a period of 90 days

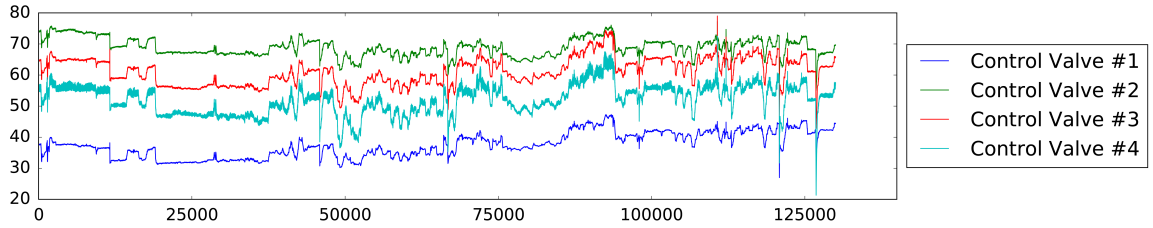


Figure 2.6: FC on the first pass has to open the control valve increasingly more over a period of 90 days, in order to keep the flow at the required level

2.2.2. Use Case 2: Manual Bypass Valve

The second use case is based on the anomalous behaviour of the same valve that caused issues in the first use case. In this case, it is visible that the manual bypass valve has been opened for periods of time, causing the output from the flow controller to the control valve to saturate. This behaviour is caused by the inability of the control valve to regulate the flow, which forces the flow controller to increase its output until it saturates.

Industry experts have not analysed this use case, therefore, its objective was to evaluate whether the model developed for the first use case is capable of predicting the opening of the manual bypass valve.

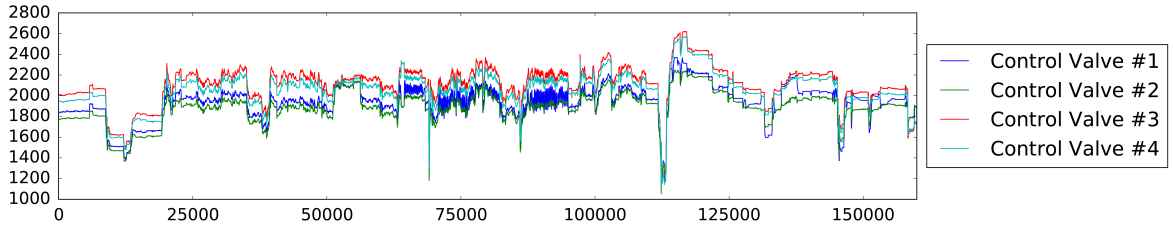


Figure 2.7: Setpoints for all FCs in F***B have the same relationship over a period of 111 days

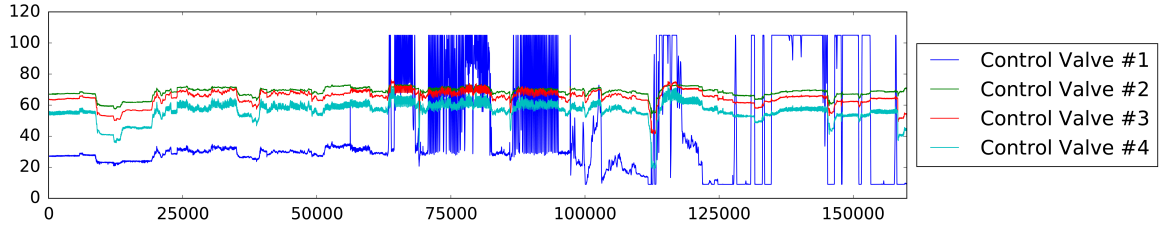


Figure 2.8: FC on the first pass has to saturate the output to the control valve while trying to regulate the flow, because the bypassed control valve does not have the desired effect on the flow

3

Machine Learning

This chapter presents the machine learning approaches that could be used for early fault prediction and predictive maintenance in petrochemical refineries. It is started with an introduction to machine learning is Section 3.1, which is followed by a discussion of high-level approaches that could be used for prediction in Section 3.2. Then, the underlying concepts of specific machine learning methods as well as their prior use cases for anomaly detection will be provided. Section 3.3 presents linear regression, which is a statistical approach for modelling that is commonly categorized under machine learning. Section 3.4 provides an overview of support vector machines, which are one of the most popular approaches for anomaly detection. Section 3.5 presents a range of artificial neural networks, which are becoming increasingly more popular. Finally, the different methods will be summarised in Section 3.6.

3.1. Basics

Machine learning is used to enable computers to learn from data rather than being explicitly programmed. The data is separated into training and testing sets or into training, testing and validation sets. The training set is used to train the model for a certain application. The testing set is used evaluate the performance of the already trained model while being presented with previously unseen data. Finally, the validation set can be used as an unbiased evaluation metric, as the structure and the parameters of the machine learning models are reconfigured based on the model's performance for the testing set. Different machine learning approaches are commonly classified into three categories:

1. *Supervised learning.* A common approach for training machine learning models is to explicitly state the correct output for any given input vector, therefore, supervised learning is based on labelled datasets. The machine learning models are trained using a collection of training examples, where each example is a pair of an input vector and an output vector. Therefore, a function is inferred from a set of input examples, which can be used to make predictions on unseen data. This approach is named supervised learning, because the correct mapping between inputs and the resulting outputs is predefined.

2. *Unsupervised learning.* Unsupervised learning deals with datasets that do not have predefined outputs for any given inputs. Hidden structure is extracted from unlabelled training data, which means that the machine learning algorithms have to extract the underlying concepts by themselves.

3. *Reinforcement learning.* Humans learn from experiments rather than by being explicitly told how to learn. Similarly, reinforcement learning strives for an automated process for learning the best approach in any given context. It is achieved by performing an action multiple times over, where a reward is given for a positive outcome and punishments for negative outcomes. The machine learning model will change its parameters based on the feedback and retry the action with a new configuration until sufficient performance is reached.

This chapter presents supervised and unsupervised machine learning approaches that can be used for early fault prediction. However, reinforcement learning methods are not discussed. Supervised learning

is separated into regression and classification problems. Regression problems, also known as function approximations, generate a function that closely approximates the target function based on the training data in a way, that the predictions are continuous. Classification problems generate a function, that is capable of identifying to which of a collection of categories a new observation belongs, therefore, making discrete predictions. Commonly used unsupervised learning approaches are clustering and anomaly detection, which separate examples into discrete groups based on their similarity.

3.2. Fault Detection

Petrochemical refineries have large amounts of process control data available, however, none of that data is labelled. Therefore, it is not possible to solve the problem of early fault prediction using classification. Furthermore, the total amount of failures for the monitored subsystem in the refinery is in the order of 10, which means that the difference between the number of training examples for normal behaviour and for abnormal behaviour is extremely high. The solution when working with such datasets is to create a model that represents the normal state of a system. This can be achieved by training the model with only the process control data for the normal operation of the refinery. Faults can be detected if the new datasets from the system result in different behaviour than the one learned by the model. However, the refinery might work correctly even if abnormal conditions take place in a component, because the closed-loop controllers are capable of handling large disturbances in the system. Therefore, it is important to filter the input dataset before the actual training phase.

Model-based fault prediction methods that use machine learning commonly fall under two categories. The first approach is to train a model to predict the real-valued outputs of a system at time t based on the prior inputs to the system at time $t-1$ or based on a sequence of prior inputs until time $t-n$. At run time, faults can be detected by comparing the predicted output of the system to the actual output of the system, as illustrated in Figure 3.1. A notification can be sent if a predefined error threshold is crossed. The difference between the prediction and the measured value can be used as a confidence metric, that enables the operators of the refinery to evaluate the notifications based on their criticality.

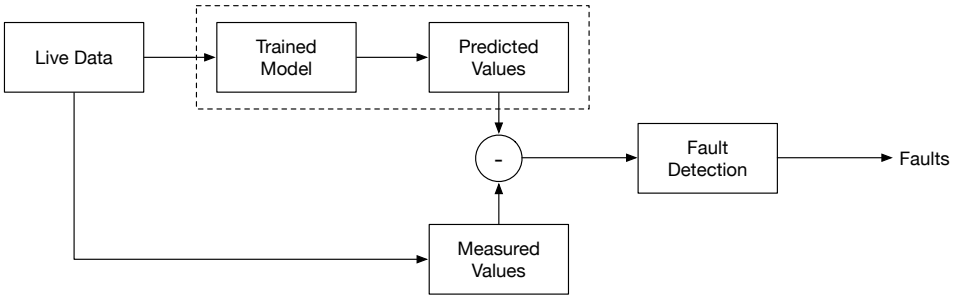


Figure 3.1: Fault detection methods with real-valued outputs

The second option is to use one-class classifiers that can detect patterns that have not been presented to the model during the training phase. Such methods are composed of a simple pipeline, as presented in Figure 3.2. It is possible to obtain a confidence metric for the prediction when using some of these methods, although the output of these methods is binary.

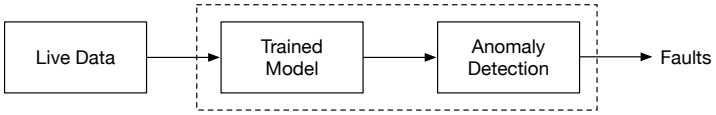


Figure 3.2: Fault detection methods with binary outputs

3.3. Linear Regression

Linear regression was the first regression analysis method that was extensively used and it still has many practical applications.

Simple linear regression is a modelling approach, where the relationship between a scalar dependent variable y and a scalar independent variable x is estimated. The approach can be extended to use multiple independent variables, which is denoted as multiple linear regression. Also, the approach can be extended to use multiple dependent variables, which is known as multivariate linear regression. [14]

Given a set of data $\{x_{i1}, ..., x_{ip}, y_i\}_{i=1}^{n}$, a linear regression model estimates the relationship between the dependent variable and the independent variables as follows [12]:

$$y_i = \boldsymbol{x_i}^T \boldsymbol{\beta} + \epsilon_i, \qquad i = 1, ..., n, \tag{3.1}$$

which is usually stacked together into a concise form as follows [15]:

$$y = \beta_0 + \boldsymbol{\beta_1 x} + \epsilon, \tag{3.2}$$

where $\boldsymbol{y}$ is the dependent variable, $\boldsymbol{x}$ is the independent variable vector, β_0 and $\boldsymbol{\beta_1}$ are parameters and ϵ is the error term.

The function between the independent variables and dependent variables can be modelled as an n-th degree polynomial in x, which is denoted as polynomial regression. It enables the model to describe nonlinear relationships between the independent variables and dependent variables, which is not possible with plain linear regression. Polynomial regression is considered to be special case of linear regression, although it fits a nonlinear model to the dataset, because it is linear as a statistical estimation problem. [14]

The linear regression model can be extended for any n-th degree polynomial regression model:

$$y = \beta_0 + \boldsymbol{\beta_1 x} + \boldsymbol{\beta_2 x^2} + ... + \boldsymbol{\beta_n x^n} + \epsilon \tag{3.3}$$

A range of methods have been designed for parameter estimation in linear regression, of which ordinary least squares, ridge regression and lasso regression are the most common ones and will be further described in the following sections.

3.3.1. Ordinary Least Squares

Ordinary least squares (OLS) is the simplest and the most common linear regression estimator. The method minimizes residual sum of squares between the observations and the predictions by the linear approximation. The mathematical notation of ordinary least squares can be written as [15]:

$$min\left\{\sum_{i=1}^{n} \left(y_i - \boldsymbol{x_i}^T \boldsymbol{\beta}\right)^2\right\} \tag{3.4}$$

However, for ordinary least squares the input features have to be linearly independent. Otherwise, the phenomenon of multicollinearity will appear, which means that one of the predictor variables can be linearly predicted using the others with great accuracy. This might cause high fluctuations in the coefficient estimates in response to small changes in the data. This redundancy in the dataset might cause overfitting to the training data, which means that the model describes random noise instead of underlying relationships. [16]

The dataset from Shell Pernis includes a wide range of independent variables that might be correlated,

therefore, the ordinary least squares method will not be further evaluated in the research. The common approaches to deal with multicollinearity include partial least squares regression, ridge regression, and principal component regression.

3.3.2. Ridge Regression

Regularization is a powerful technique used to overcome overfitting and ill-posed problems by introducing additional parameters into the loss function of the estimator. The most common regularization method is ridge regression, also known as Tikhonov regularization. It is not susceptible to multicollinearity, therefore, it can be used with correlated features. Furthermore, it improves the performance of linear approximation when using a large number of input features. [16]

Ridge regression penalizes the magnitude of coefficients of the features while minimizing the error between the observations and the predictions by the linear approximation [15]:

$$min\left\{\sum_{i=1}^{n}\left(y_i - \boldsymbol{x_i}^T\boldsymbol{\beta}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2\right\}, \tag{3.5}$$

where λ is the regularization parameter that controls the amount of coefficient shrinkage.

Therefore, ridge regression performs L_2 regularization, which means that it adds the sum of squares of coefficients to the loss function. Large coefficients contribute to overfitting, because it can boost the effect of noise and insignificant features to the prediction. The value of λ has a significant effect on the coefficients β:

1. $\lambda = 0$. The coefficients will be equal to linear regression with ordinary least squares.
2. $0 < \lambda < \infty$. The coefficients will be greater than 0 but smaller than for linear regression with ordinary least squares.
3. $\lambda = \infty$. The coefficients will be zero.

Bender, et al. (2016) [23] have already evaluated the applicability of ridge regression for early fault detection in petrochemical refineries, specifically the Shell Pernis refinery in Rotterdam. A range of linear regression methods was evaluated, but ridge regression was found to provide superior results when compared to others.

3.3.3. Lasso Regression

Lasso regression, which stands for least absolute shrinkage and selection operator, is similar to ridge regression, but instead of L_2 regularization it performs L_1 regularization. Therefore, it penalizes the loss function by the absolute value of the weights [15]:

$$min\left\{\sum_{i=1}^{n}\left(y_i - \boldsymbol{x_i}^T\boldsymbol{\beta}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j|\right\}, \tag{3.6}$$

where λ is the regularization parameter that controls the amount of coefficient shrinkage.

The key difference between L_1 and L_2 regularization is that L_1 regularization performs feature selection in addition to coefficient shrinkage. This is achieved by the fact that a significant number of

coefficients become close to zero, therefore, neglecting features that have little effect on the prediction. [11]

The applicability of lasso regression for petrochemical refineries has been evaluated by Bender, et al. (2016) [23], and was found to perform worse than ridge regression. Furthermore, the inherent property of lasso regression to perform filter selection might be in contrast with the goal of this research, since correlating features in the training set might behave differently when making predictions for new data.

3.4. Support Vector Machines

Support Vector Machines (SVMs) are supervised machine learning models that can be used for classification as well as regression problems. However, one-class SVMs are most commonly used for anomaly detection. When using SVMs for the classification of linearly separable classes, its goal is to devise a hyperplane that correctly classifies all training examples. For linearly separable two-class problems with classes ω_0, ω_1, the hyperplane is defined as [54]:

$$g(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0 = 0, \tag{3.7}$$

where $\boldsymbol{x}$ is the feature vector and $\boldsymbol{w}$ is the weight vector. However, the obtained hyperplane is not unique, therefore, it can be biased towards either of the classes. In order to obtain a robust classification method, it would be desirable to design the hyperplane in a way that maximizes the margin from the hyperplane to the nearest training examples from both classes. It enables the classifier to provide more robust predictions for new data, as the data points can be positioned slightly towards the other class without providing erroneous predictions, as the margin from the hyperplane towards both classes has been maximized. In order to achieve this, $\boldsymbol{w} + w_0$ have to be scaled in a way that $g(\boldsymbol{x})$ for the nearest points in ω_0, ω_1 is 1 and -1, respectively. Therefore, for every $\boldsymbol{x}_i$ the corresponding class label will be y_i. Mathematically, this results in [54]:

$$min\left\{J(\boldsymbol{w}, w_0) \equiv \frac{1}{2}||\boldsymbol{w}||^2\right\} \text{ subject to } y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + w_0\right) \geq 1 \ \forall i \tag{3.8}$$

The selection of the best hyperplane based on the maximized margin between the hyperplane and the nearest data points from each class is the main distinguishing factor of SVMs from other linear classifiers. The small number of data points that specify the hyperplane are known as support vectors. Linear classifiers, however, are limited in the scope of problems that can be solved using them. Therefore, a method known as the kernel trick has been devised to enable the usage of SVMs to perform classification for non-linear problems. This is achieved by mapping the input feature space into a higher-dimensional space, where the classes can be separated by a hyperplane [54]:

$$\boldsymbol{x} \in \mathbb{R}^l \rightarrow \boldsymbol{y} \in \mathbb{R}^k \tag{3.9}$$

Therefore, non-linear class boundaries in the original space $\mathbb{R}^l$ can be separated by a hyperplane in a higher-dimensional space $\mathbb{R}^k$, as the kernel function is non-linear. SVMs are widely used as they generalize well, provide a unique solution and can model almost any dataset if the kernel function is correctly created [49].

A type of SVMs, the one-class SVM, is one of the most widely used methods for anomaly detection, as they are used to model the normal data points which can then be used to detect anomalies amongst novel data points. A variety of one-class SVMs has been devised that can be based on hyperplanes, hyperspheres or hyperellipsoids, as illustrated in Figure 3.3. Schölkopf, et al. (1999) [21] have devised a one-class SVM that separates the normal data points from the origin using a hyperplane in a high-dimensional space $\mathbb{R}^k$ with the maximal margin from the origin. Then, the anomalous data points will be mapped to the half space that contains the origin. Tax, et al. (2004) [53] proposed a method that encloses the normal data points in a high-dimensional space $\mathbb{R}^k$ with a hypersphere that has the

smallest radius. Then, the data points that are outside of the hypersphere can be detected as anomalies. Similarly, Wang, et al. (2006) [25] have proposed a hyperellipsoid based approach to detect anomalies.

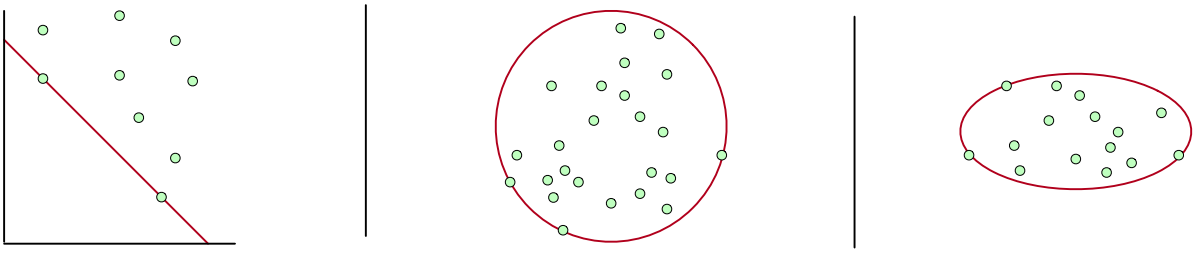


Figure 3.3: a) Hyperplane-based one-class SVM b) Hypersphere-based one-class SVM c) Hyperellipsoid-based one-class SVM

One-class SVMs have been used for anomaly detection in a wide variety of domains [32, 46, 48]. Garcia-Font et al. (2016) [55] compared four anomaly detection methods based on real-life data from wireless sensor networks and concluded, that one-class SVMs provide the best results. However, recent approaches commonly use one-class SVMs in combination with other methods. Marti, et al. (2015) [38] have proposed an anomaly detection method for petrochemical refineries, that combines a custom segmentation algorithm with one-class SVMs. The proposed method outperformed plain one-class SVMs and more impressively, was deployed by a petroleum industry conglomerate in Brazil. Furthermore, one-class SVMs have already been used for anomaly detection in petrochemical installations internally by Shell.

Although one-class SVMs have been successfully used for anomaly detection, the best results can be obtained with relatively small datasets that have a large number of features. However, their computational complexity increases quadratically with the number of training examples, although optimizers that reduce the computational complexity have been devised. Therefore, the number of research articles that evaluate one-class SVMs on large datasets is limited.

3.5. Artificial Neural Networks

An artificial neural network (ANN), commonly referred to as neural network (NN), is a computational model inspired by the human brain that is composed of neurons. The power of neural networks is derived from its massively parallel construction and from its ability to generalize. Neural networks feature a range of appealing properties [27]:

1. *Nonlinearity.* Neurons can be either linear or nonlinear. Therefore, they can be used for problems, which are fundamentally nonlinear.

2. *Input-Output Mapping.* ANNs are capable of mapping the inputs of the training set to their respective outputs by modifying the weights and biases of the network until the difference between the desired response and the actual response reaches a steady state.

3. *Adaptivity.* Neural networks trained for certain applications can be easily retrained for new applications, if the differences are minor.

4. *Evidential Response.* A measure of confidence can be provided for classification problems.

5. *Contextual Information.* Neural networks deal with contextual information naturally, since all neurons are potentially influenced by all other neurons.

6. *Fault Tolerance.* ANNs, that are implemented in hardware, have the potential for robust computation, because damaged neurons or connections degrade the performance of the network instead of causing failures. Catastrophic failures can be caused only if there is extensive damage to the network.

7. *VLSI Implementation.* ANNs are well suited for hardware implementation.

8. *Uniformity of Analysis and Design.* The same notations and theories can be used in all application domains that use neural networks.

9. *Neurobiological Analogy.* ANNs have enormous potential, since the human brain is a living proof of a fast, powerful and energy-efficient parallel information processor that is capable of learning.

A neuron is the basic information processing unit in neural networks, that is composed of connections, a bias, an adder and an activation function. Mathematically, it can be represented with a pair of equations [27]:

$$u_k = \sum_{j=1}^{m} \omega_{kj} x_j \tag{3.10}$$

and

$$y_k = \varphi\left(u_k + b_k\right), \tag{3.11}$$

where $\omega_{k1} \ldots \omega_{km}$ are the weights; $x_1 \ldots x_m$ are the inputs; u_k is the output of the adder; b_k is the bias; $\varphi(\cdot)$ is the activation function; and y_k is the output of the neuron.

The output of a neuron is defined by the activation function $\varphi(\cdot)$, which performs a mathematical function on its input. A variety of activation functions are used in practice [2], the most common ones being:

1. *Sigmoid.* Computes $1/(1+e^{-x})$, therefore, squashing a real-valued number between 0 and 1. Its usage has seen a downfall, because its outputs are not zero-centred and it is prone to saturation, which reduces its learning capacity [3].

2. $tanh$. Computes $tan(x)$, therefore, squashing a real-valued number between -1 and 1. $tanh$ saturates similarly to sigmoid, but it is more widely used because its output is zero-centred [3].

3. *ReLU.* Computes $max\,(0, x)$, therefore, clipping the negative inputs to zero. ReLU converges faster and is computationally less expensive than sigmoid and $tanh$, but the units can irreversibly die during training, which means that a large number of neurons never affects the output of the neural network.

4. *Dropout.* Dropout is a regularization technique for neural networks, that blocks the passage of values from the input of an activation function to its output with a predefined probability.

The neurons in a neural network are structured into layers, where every layer might contain one or more neurons. There are three types of layers: input layers that are connected to the inputs of the model; hidden layers that are not visible in the training set; output layers that present the output of the model. The overall number of layers is referenced as the depth of the model, while the number of neurons that are in a layer is referred to as the width of the model. ANNs are categorized into feedforward and recurrent neural networks based on the connections between the layers. Feedforward neural networks contain zero or more hidden layers, where all of the outgoing connections from layer n will reach layer $n+1$. Recurrent neural networks feature an additional feedback loop that enables it to learn from sequences instead of discrete training examples. [27]

3.5.1. Deep Feedforward Neural Networks

Deep neural networks (DNNs) are feedforward neural networks that contain multiple hidden layers, as illustrated in Figure 3.4. In general, feedforward neural networks approximate a function $f(\cdot)$, which maps the input vector x to the output vector y [28]. This means, that the training set specifies the behaviour of the output layer in response to values in the input layer. The behaviour of the hidden layers is not defined by the training set and has to be configured by the learning algorithm in order to find the best approximation of $f(\cdot)$.

DNNs can be used for both classification and regression problems. They have been used for early fault detection in industrial systems using only unlabelled sensor data by Tjernberg, et al. (2015) [22]. The DNN is trained using the normal operating data to predict a measured parameter based on a wide range of measured features. Then, the model can be used to make predictions of the measured parameters during run time, which can be compared to the actual measurements of that parameter. A notification can be sent to system operators if a difference threshold has been crossed. The experiments indicated, that the proposed approach was able to provide an early notification of a potential issue almost a week earlier than a vibration-based condition monitoring system. The anomaly detection pipeline used by Tjernberg, et al. is similar to many solutions using linear regression as well as recurrent neural networks, the main differentiator between those approaches is the machine learning method that is used in the pipeline.

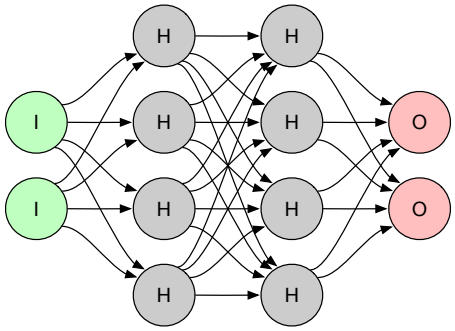


Figure 3.4: Conceptual design of deep neural networks

3.5.2. Autoencoders

Autoencoders (AEs) are feedforward neural networks designed for unsupervised learning of efficient codings. Simple autoencoders are similar to feedforward neural networks, having an input layer, one or more hidden layers and an output layer. The distinguishing differentiator of autoencoders is the number of nodes in the output layer, which is equal to the number of nodes in the input layer. Therefore, it learns an approximation to the identity function by reconstructing the input vector at the output as illustrated in Figure 3.5. The mapping of the inputs to the intermediate representation can be described as encoding and the mapping of the intermediate representation to the outputs as decoding. [4]

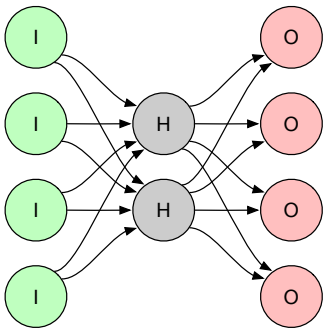


Figure 3.5: Conceptual design of autoencoders

The resulting method is lossy, which means that the reconstructed outputs will be degraded when compared to the raw inputs. It can learn efficient representations automatically from the data, but this means that it is data-specific and will only be able to compress similar patterns to what it has been trained on. Furthermore, autoencoders can efficiently encode the inputs only if there is structure in the data, completely random inputs will not result in low-dimensional representations. Therefore, it can be used for data compression, but it is more commonly used for data denoising and dimensionality reduction. It has to be noted that simple autoencoders with a single hidden sigmoid layer learn similar representations to principal component analysis, a method that is widely used for data preprocessing. [4]

A variety of autoencoders exists, one of the most common ones being denoising autoencoder. It obtains a robust representation by introducing noise into the input dataset while preserving the correct dataset at the output. This method is named denoising, since it removes noise from the input dataset to obtain a clean reproduction of the inputs. [4]

Autoencoders are trained by finding network weights and biases that minimize the reconstruction error between the input vector $\boldsymbol{x}$ and its reconstruction at the output vector $\overline{\boldsymbol{x}}$ [49]:

$$min\left\{||\boldsymbol{x}-\overline{\boldsymbol{x}}||^2\right\} \tag{3.12}$$

Autoencoders can be used to construct end-to-end anomaly detection approaches by evaluating the reconstruction error of new observations. A threshold τ can be set for the reconstruction error based on statistical metrics of the historical reconstruction errors for $x_i \in X$, where $i = 1, ..., m$ [49]. This approach is the simplest method of anomaly detection when using autoencoders. Lyudchik, et al. (2016) [39] have used this approach to perform outlier detection on images of digits. Autoencoders were trained with images of all digits in the range 0…9, except for 7. Experiments with multiple autoencoder configurations were performed, where several approaches yielded promising results. Nevertheless, the number 7 was reconstructed with relatively high probability, although it had been excluded from the training dataset.

Additionally, autoencoders can be used as an unsupervised feature extraction method for anomaly detection. One-class SVMs have been used extensively for anomaly detection in low-dimensional spaces, but it is inefficient for high-dimensional problems. Therefore, a hybrid model of autoencoders and one-class SVMs has been proposed by Erfani, et al. (2016) [49], where autoencoders are used for feature extraction and SVMs for anomaly detection. The proposed hybrid model was found to reduce the training time by a factor of 3 and the testing time by a factor of 1000, when compared to solutions using deep autoencoders. The implementation has been evaluated across a range of real-life datasets from the UCI Machine Learning Repository, which showed that the difference in the accuracy between the proposed approach and deep autoencoders is negligible. Furthermore, the achieved performance was increased by 20% when compared to one-class SVMs without autoencoders.

Labelling the datasets from petrochemical plants is time consuming and expensive, because the processes have to be analysed by industry experts. Nevertheless, a small range of labelled examples might be available for most normal and abnormal use cases. A supervised learning approach has been devised for petrochemical plants [43] that is capable of utilizing these small labelled datasets to train a classification method in combination with large unlabelled datasets. A stacked denoising autoencoder (SDAE) has been used as the unsupervised pre-training method, the output of which is fed to a supervised classification network based on the softmax function. The supervised learning stage is used to reduce the training error by utilizing a small amount of labelled data. Then, an active learning method based on the Best vs. Second Best criterion (BvSB) and Lowest False Positive criterion (LFP) is used to further train the model in an active manner. In general, active learning is used to learn a function that is capable of improving the performance of a model while utilizing as little labelled data as possible.

3.5.3. Recurrent Neural Networks

Feedforward neural networks allow information to pass only from their inputs to their outputs, i.e. all training examples are independent from each other. This approach is sufficient for problem domains that deal with static data, but it does not work well with sequences of data. This limits the applicability of DNNs for a wide range of problems, e.g. natural language processing, video classification and machine translation. Recurrent neural networks (RNNs) [27] introduce feedback loops into the structure of the network in order to allow information to persist, therefore, making it possible to learn from sequences of data.

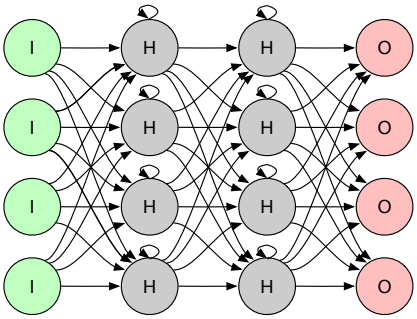


Figure 3.6: Conceptual design of recurrent neural networks

A single-layer RNN is displayed in Figure 3.7, which features a feedback loop in the hidden layer. Although the feedback loops make RNNs mysterious when compared to DNNs, it is possible to unroll them to a form that is similar to DNNs. Therefore, RNNs can be thought of as n copies of interconnected DNNs with the exact same network configuration and weights at every step, where n is the length of RNN memory [42]. It has to be noted, that vertical arrays of neurons are presented as a single unit for brevity.

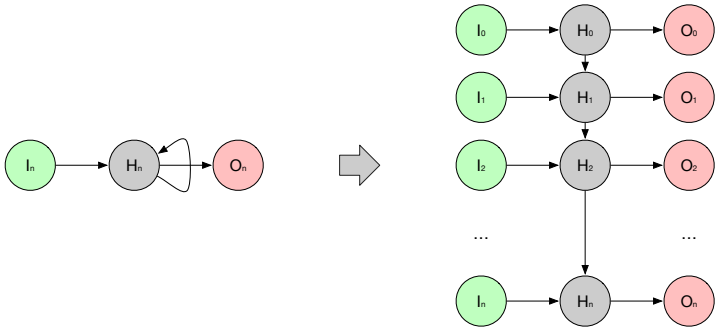


Figure 3.7: Unrolling of RNNs

Unfortunately, this simple RNN structure, also known as vanilla RNN, is not capable of learning from long-term dependencies that influence the output. The derivatives of sigmoid and $tanh$ functions become zero when saturated, which means that their gradient is zero and they drive other gradients in previous layers towards zero. Therefore, the gradient values will shrink exponentially fast and can vanish completely after only a small number of steps [27]. This issue is known as vanishing gradient problem, which makes them rarely used in practice.

Vanishing gradient problem has been overcome by using more complex RNN unit architectures, the most common ones being the long short-term memory (LSTM) and gated recurrent unit (GRU). LSTMs were introduced in 1997 and they have been used successfully in new products, e.g. Google Allo, Google Translate, Amazon Alexa, Apple Siri [13]. Vanilla RNNs are constructed of a single neural network layer, such as $tanh$. LSTMs introduce 3 more layers that interact with each other as displayed in Figure 3.8, where they are marked with yellow squares. Every line represents an entire vector of values that are carried between nodes. Red circles perform pointwise operations on the vectors, i.e. multiplication and

summation. LSTMs persist information by making modifications to the cell state, which is represented by the horizontal line in the top of the cell, where information can move through the entire LSTM chain without large modifications. Outdated information can be removed from the cell state using the $\times$ operation on it, where the amount of discarded information is controlled by a sigmoid neural network layer. In total, there are three sigmoid layers that can control the amount of information that is passed through by setting its output between 0 and 1. Then, the LSTM cell will decide what new information to store in the cell state. A vector of new values is created by the $tanh$ layer, the output of which is multiplied by the output of the sigmoid layer. This enables the LSTM cell to select which values of the vector created by the $tanh$ layer should be allowed to pass through to the cell state. Then, the new vector will actually be added to the cell state with the $+$ operation. Finally, the LSTM cell modifies the cell state in order to generate the final output. This is achieved by squashing the values between -1 and 1 using the $tanh$ operation followed by a sigmoid layer that selects which values to output. [42]

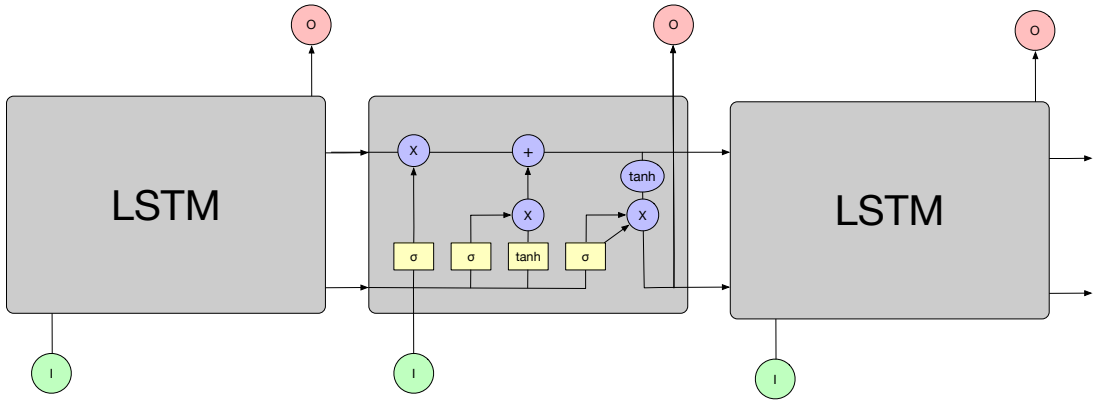


Figure 3.8: Internal construction of LSTM cells

A variety of LSTM modifications have been published, e.g. LSTM by Gers & Schmidhuber (2000), GRU by Cho, et al. (2014) and Depth Gated RNN by Yao, et al. (2015) [35]. GRUs have become increasingly popular in recent years, since they simplify the unit construction when compared to LSTMs.

Greff, et al. (2015) [33] and Jozefowicz, et al (2015) [45] have evaluated thousands of RNN architectures to find the best approach. They failed to find solutions that continuously outperform standard LSTMs and GRUs, although different architectures did perform better for some use cases. The most important blocks in LSTMs were found to be the forget gate and the output activation. Finally, they found that the performance of LSTMs is mainly affected by the learning rate and network size hyperparameters. Chung, et al. (2014) [29] have compared standard LSTM and GRU implementations to see, whether one is superior to the other. GRU's simplified construction should improve its training time while the added complexity of LSTMs might increase its representative power. However, they were not able to conclude whether one is superior to the other for most use cases.

Different RNN configurations are illustrated in Figure 3.9. Time series predictive maintenance applications can be either many to one or many to many, as the input to the prediction always has to be a sequence. However, the output could be either a single value or a sequence. Single output configurations can easily be integrated with the fault detection methods described in Section 3.2. Configurations that predict a sequence can be used to detect the anomalies in the same way as many to one configures. However, sequence predictions can be used to provide the operators of the refinery with the future trends of a parameter in addition to the general notification, as this enables the operators to evaluate the criticality of the notification based on the predicted changes in the near future.

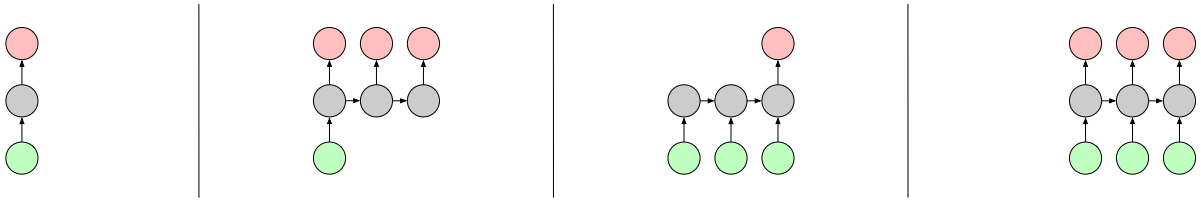


Figure 3.9: a) *One to one*. A single output is predicted based on a single input, i.e. processing without RNNs b) *One to many*. An output sequence is predicted based on a single input, e.g. describing the contents of an image c) *Many to one*. An output value is predicted based on an input sequence, e.g. predicting the next time series value based on prior values d) *Many to many*. An output sequence is created predicted on an input sequence, e.g. predicting a sequence of next time series values based on prior values

At the start of this research project in 2016, relatively few articles that had used RNNs for anomaly detection had been published. Nevertheless, this situation has changed in the meantime: Bontemps, et al. (2017) [37] have designed an RNN-based anomaly detection method for intrusion detection based on time series data, where an intrusion is detected when the difference between the predicted value and the actual value crosses a threshold; O'Shea, et al. (2016) performed novelty detection for radio signals [52]; Sherry, et al. (2017) used RNNs for anomaly detection in aircraft data [40]; Zhao, et al. (2017) [47] used RNNs in combination with CNNs for tool wear monitoring.

RNNs are especially well-suited for time series data where new measurements are dependent on the prior states of the system. Petrochemical refineries are composed of a series of valves, pumps and distillation columns among others, which all influence the behaviour of each other. Furthermore, every control valve is a PID controller, therefore, RNNs should be capable of modelling the behaviour of control valves better than methods that are based on static data. However, RNNs with a long history require orders of magnitude longer training times when compared to shallow methods, therefore, it has to be evaluated whether the potential gains in the prediction quality are worth the additional computational as well as implementational complexity.

3.5.4. Convolutional Neural Networks

DNNs have been used for a wide range of applications that take images as their inputs, but they do not scale well for large images. A small 32×32 pixel RGB image that is used by the CIFAR-10 dataset requires $32 \times 32 \times 3 = 3,072$ weights for every neuron, if fully-connected neurons are used. The amount of weights per neuron increases to $1920 \times 1080 \times 3 = 6,220,800$ for the commonly used 1080p video format. Practical applications require a large amount of neurons across multiple layers, therefore, the amount of parameters increases quickly. It makes DNNs wasteful, computationally expensive and prone to overfitting when used with large images [5].

Convolutional neural networks (CNNs) [5] are a subtype of feedforward neural networks, that are specifically designed to process images as their inputs. This presumption has reduced the applicability of CNNs for other problem domains, but it enables them to achieve superior results for image processing when compared to DNNs. In general, CNNs are very similar to the standard feedforward neural networks described in Section 3.5.1: CNNs are composed of neurons that have weights and biases; CNNs perform linear or non-linear transformations on its inputs; CNNs have a loss function on their last layer. The core difference between CNNs and DNNs is, that the neurons in CNNs will only be connected to a small region of the prior layer instead of being connected in a fully-connected manner. Furthermore, CNNs use a different layer structure than DNNs.

CNN architectures are commonly composed of an input layer that holds the raw pixel values, convolutional and pooling layers that are specific to CNNs and fully-connected layers that compute the class scores that are passed to the output, as illustrated in Figure 3.10.

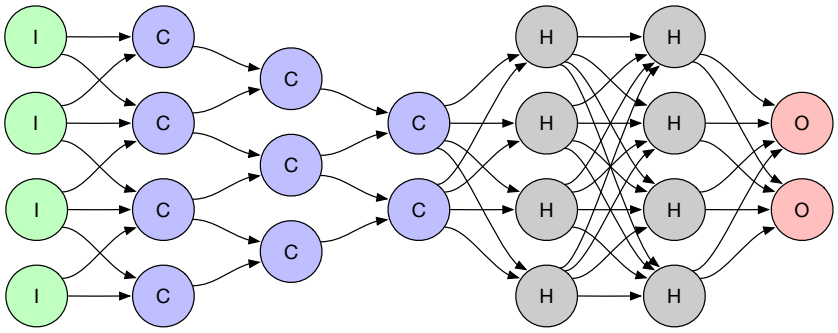


Figure 3.10: Conceptual design of convolutional neural networks

DNN layers are constructed of a vector of neurons, that are connected to all of the neurons in the prior layer. CNN layers are reconfigured to transform 3D inputs to 3D outputs, where the dimensions of a layer are described as the width, height and depth. CNNs are commonly constructed of three different layer types: convolutional and pooling layers that are specific to CNNs and fully-connected layers that are used in regular DNNs.

Convolutional layers are at the heart of CNNs. CNNs are composed of a set of filters, where every filter has small dimensions along width and height, but extends through the full depth of the input [5]. The behaviour of convolutional layers is illustrated in Figure 3.11. Every filter in the convolutional layer is connected to a section of the input image across all RGB layers. In this example, filters look at a 2×2 region in the input image, therefore, using $2 \times 2 \times 3 = 12$ weights. Then, the filter will be slided across the input image, resulting in a 2-dimensional activation map that represents the values of a filter at any given spatial position. The process will be repeated for all filters, that represent different aspects of the input image.

Convolutional layers are controlled by three hyperparameters: depth represents the number of filters in a layer; stride is the step size when sliding the window across the input; zero-padding controls the number of additional cells with a value of zero, that are used to surround the input volume [5].

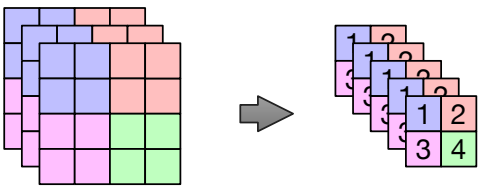


Figure 3.11: Convolutional layer with 5 filters applied to a 4×4 RGB image without zero-padding and with a stride of 2

In addition to the convolutional layers, it is common to use pooling layers in CNNs [5]. Pooling layers reduce the resolution (width and height) of the filters, therefore, reducing the amount of parameters used in the network. Effectively, this controls overfitting and reduces the computational complexity of the problem. The dimensions are reduced by using mathematical operations, such as max, $average$ and $L_2 - norm$. The behaviour of the max operation is illustrated in Figure 3.12. It has to be noted, that the depth of the volume, i.e. the number of filters, stays constant when pooling is applied. CNNs are usually constructed of multiple interleaved convolutional and pooling layers, which are then followed by fully-connected layers.

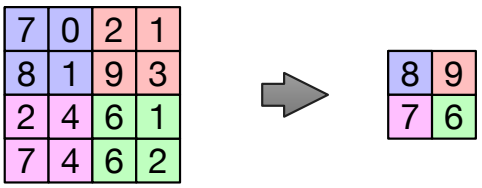


Figure 3.12: Example of a pooling layer performing the max operation on a single 4×4 filter

Effective applications have been created using neural networks, yet the developers rarely understand what and how a network learns. One of the main benefits of CNNs when compared to RNNs and regular DNNs is, that tools capable of visualizing CNNs, e.g. Deep Visualization Toolbox (2015) [1], are readily available. In general, a variety of methods exist that can be used to visualize CNNs [6]:

1. *Layer Activations.* The specific neurons that an image triggers can be visualized by passing a set of images via the network one by one, while plotting the activation maps of all filters at every layer. Differences between the activation maps of a set of images can then be used to search for patterns between images.

2. *Weight Visualization.* Overfitting and the lack of training epochs can be detected by visualizing the weights of a filter, because well-trained filters map to smooth images. Therefore, sharp patterns can indicate that the network is overfitting or hasn't been trained for long enough.

3. *Maximally Activating Images.* Learned features of a single neuron can be visualized by feeding a large set of images through the network, while tracking which images maximally activate a single neuron in the network.

4. *Code Embeddings with t-SNE.* Images, that a CNN thinks are similar can be visualized using the t-SNE embedding of a set of images based on their representation in the last hidden layer of a CNN.

5. *Modified Images.* CNNs can be analysed as a whole by systematically setting blocks of pixels in an image to zero. Then, these images can be presented to a CNN to see how it classifies them and with what probability. Important parts of an image that are learned by the network can be detected, if a modified image causes the CNN to produce erroneous predictions. Additionally, it makes it possible to verify whether a network has actually learned a feature that is known to have a significant contribution to the prediction, e.g. a bad network might detect vehicles with the same probability even if the wheels of a vehicle are removed from the input image.

At first glance, the connection between time series anomaly detection and CNNs seems to be far-fetched, yet such solutions exist. CNNs have been used for heart disease detection in a commercial product, the AliveCor Kardia [51]. It is clear, that AliveCor has mapped time series data to multiple horizontal images that are combined together into a single image that is passed to the neural network. Nevertheless, the specifics of the product have not been published. Additionally, Wang, et al. (2016) [34] have used two convolutional and two pooling layers for processing high-dimensional physiological signals, the output of which is mapped into multivariate Gaussian anomaly detection model. Interestingly, the CNNs are trained in an unsupervised manner using AEs. Although CNNs are an interesting approach for anomaly detection in time series data, they are still rarely used in practice when compared to other methods described in this chapter. Furthermore, the mapping of high-dimensional time series data to images remains complex and the effect of different encoding has not been analysed.

3.6. Summary

A brief summary of the analysed methods is presented in Table 3.1. RNNs were considered to be the most promising machine learning method for the use cases at Shell Pernis. The main advantage of RNNs when compared to other methods is, that it is capable of looking at time series data. It should yield the best results, because petrochemical refineries are composed of a sequence of structure, where the first components in the structure affect the subsequent components. Additionally, DNNs and ridge regression were implemented to provide a baseline method for comparison. These methods are well-suited as the baseline methods for RNNs, because the output of all of these methods is obtained using the same fault detection mechanism that enables easy comparison. Furthermore, ridge regression was already found to provide good predictive capabilities at Shell Pernis by Bender, et al.

Although CNNs can be used for time series anomaly detection, they were not chosen due to the complexity of mapping the time series data to images. Furthermore, their performance cannot be directly compared to DNNs and ridge regression, which limits the comparability of the implemented methods.

SVMs and AEs were not chosen for evaluation in this research project, as these methods had already been evaluated by Shell. Therefore, the added value from the evaluation of those methods would have been limited.

Approach	Difficulty	Memory Usage	Time Series	Fault Detection
OLS Regression	+	+	N	Real-Valued
Ridge Regression	+	+	N	Real-Valued
Lasso Regression	+	+	N	Real-Valued
One-Class SVM	++	++	N	Binary
DNN	+++	+++	N	Real-Valued
AE	+++	+++	N	Binary
RNN	++++	++++	Y	Real-Valued
CNN	+++++	+++	Y	Binary

Table 3.1: Summary of the machine learning methods for early fault prediction

4

Scalability

Recent research has shown that large deep learning models can frequently outperform smaller models. However, the training time of deep learning models with billions of parameters on a single CPU has become too large for many practical applications. Novel hardware architectures as well as parallelism strategies have been devised to overcome these issues, as described in this chapter. Section 4.1 presents the different types of computational nodes that are commonly used to speed up the training as well as the inference phase of machine learning. Section 4.2 discusses model parallelism, data parallelism, and distributed training.

4.1. Specialised Hardware

4.1.1. GPUs

Central processing units (CPUs) are well-suited for sequential computations, however, they provide poor performance for inherently parallel problems. This limitation has spread the usage of graphics processing units (GPUs) for moderately sized machine learning problems in recent years. CPUs are comprised of a small number of complex cores that are specifically designed for serial processing, while GPUs are comprised of a huge number of simpler cores that are designed for parallel processing. Novel deep learning methods are based on matrix multiplications as well as other operations that can easily performed in parallel on GPUs. This enables data scientists to reduce the execution time for the training as well as the inference phase of machine learning. However, GPUs use a separate memory from CPUs that is usually less than 12GB. Therefore, larger models have to be either reduced in size or distributed across a larger number of GPUs. Furthermore, data transfers between GPUs and CPUs as well as between GPUs is slow. These issues have forced researchers to devise parallelisation methods as well as application-specific integrated circuits (ASICs), as described further in this chapter.

4.1.2. ASICs

GPUs have been at the heart of machine learning infrastructure for almost a decade, however, numerous machine learning accelerators and ASICs will be introduced in the next years that promise better performance and efficiency, e.g. the Nervana Engine that will be integrated into Intel's CPUs [41] and AI accelerators built directly into ARM chips [56]. However, the most prominent solution at the time of writing is the Tensor Processing Unit (TPU) developed by Google.

Google announced its first generation TPU in 2016 [31], which was found to improve the performance by 15-30 times when compared to best CPUs and GPUs. However, it was only designed for inference, i.e. using a trained model to perform predictions on new data, instead of training the model in the first place. This has changed with the introduction of second generation TPUs that were announced earlier in 2017 and will be available via Google Compute Engine. Up to 64 TPUs can be stacked together into a TPU pod with custom high-speed networks for data transfer, resulting in a staggering 11.5 PFLOPS

per pod [26]. Google states, that they were able to train a model with the same accuracy on 8 TPUs instead of 32 of the best GPUs, while reducing the training time from a full day to a single afternoon. Nevertheless, it is wise to be slightly sceptical towards these results, because the chosen application that is benchmarked as well as the structure of the network can be selected in a way that makes the performance difference between TPUs and GPUs the largest. Furthermore, the difference between the second generation TPU and the newly released NVIDIA Tesla V100 is much smaller than these results indicate, since the V100 reaches 120 TFLOPS as described in Section 4.1.1.

Google has already established a strong presence in the machine learning communities since TensorFlow, its machine learning framework, has become the most popular open-source machine learning framework. However, Google does not have a strong position in the cloud service market, as Amazon is the largest cloud service provider by far [44]. With the introduction of TPUs, Google can control the chips, cloud service as well as the machine learning library used. Therefore, it should be possible for Google to optimize the entire stack in a way that results in greater performance or cost reductions when compared to competition. As TPUs can only be used with TensorFlow via Google Compute Engine, it should enable Google to reinforce its position as the supplier of the most popular framework as well as increase its cloud market share.

The details of the second generation TPU have not been published, however, the architecture of the first generation TPUs was recently published by Jouppi, et al. (2017) [30]. Instead of integrating the machine learning accelerator with a CPU, as Intel and ARM are doing, it was designed as a coprocessor that can easily be deployed into already existing servers. While CPUs are scalar processors and GPUs vector processors, TPUs are built as matrix processors. At the core of TPUs is the Matrix Multiply Unit, which performs 256×256 multiplications on 8-bit integers, as illustrated in Figure 4.1. Although this level of precision is sufficient for inference, it is not suitable for training and therefore the second generation TPUs process floating point numbers. The Matrix Multiply Unit is based on systolic execution, where the inputs and weights are loaded from two separate directions and the multiplication and accumulation operations move through the matrix as a diagonal wave. The outputs of the matrix multiplication are passed through non-linear activation functions as well as pooling, if required. The intermediate results as well as the inputs are stored in the Unified Buffer, while the weights are stored in external DDR memory.

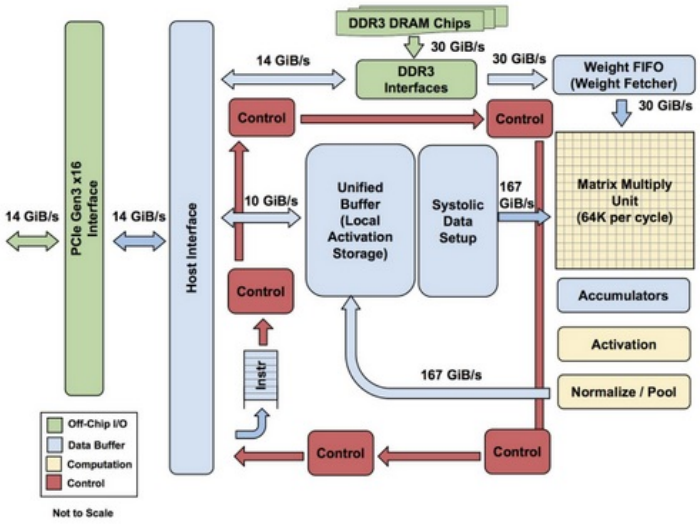


Figure 4.1: Block diagram of the first generation TPUs [30]

TPUs are controlled by the CPU over a PCIe bus using a total of 12 Complex Instruction Set Computer (CISC) instructions, however, only 5 of those instructions have been published [30]:

1. *Read_Host_Memory.* Read from CPU memory to Unified Buffer.
2. *Read_Weights.* Read from DDR memory to weight FIFO.

3. *MatrixMultiply/Convolve.* Perform a matrix multiplication or convolution with the inputs and weights.

4. *Activate.* Perform non-linear activations.

5. *Write_Host_Memory.* Write from Unified Buffer to CPU memory.

One of the main benefits of the simple TPU architecture is its power efficiency. TPUs do not have common structures found in CPUs and GPUs, e.g. caches, multithreading and multiprocessing among others, which increases their performance to power ratio. The first generation TPUs achieve 17-34 times better performance to Watt ratios than Haswell CPUs and 14-16 times better performance to Watt ratios than the K80 GPUs, however, the first generation TPUs are only capable of inference while CPUs and GPUs can also be used for training [30]. Therefore, the performance to power ratio will most probably decrease for the second generation TPUs that also feature model training. Nevertheless, even slightly smaller gains in the efficiency might enable Google to significantly reduce the total cost of ownership for their compute cloud.

4.2. Parallelisation Strategy

Selecting the best hardware platform for training machine learning models has a significant impact on the training time, yet using a single compute node poses fundamental limits on the achievable speed-up. Therefore, recent research has focused on finding solutions for training machine learning models on thousands of nodes in parallel.

4.2.1. Model Parallelism

Model parallelism tackles scalability by dividing the model between multiple compute nodes, as shown in Figure 4.2. Every node computes and stores only the parameters that belong to the section of the model allocated to it. However, some connections between neurons cross the node barriers and the state of these connections has to be transmitted between node, as illustrated with thick lines in Figure 4.2. Therefore, models that are structured to have local connectivities instead of fully-connected structures tend to scale better, because the communication overhead between nodes is smaller. Models that require significant computational power or have a large number of parameters tend to benefit more from model parallelisation. The communication overhead between the nodes might reduce the gains from model parallelisation, if the computational requirements or the number of parameters is low.

The performance of distributed training might further be reduced due to the differences in the computation times between nodes, causing many nodes to wait until the slowest node finishes a phase of the computation. However, the average difference is close to zero if the model is parallelised between multiple nodes, where each node is composed on multiple cores.

The key differentiator of model parallelism from other methods analysed in this section is its capability to train networks that do not fit into the memory of a single compute node. Although GPU memory has more than doubled in recent years, it is still not sufficient to train models that feature billions of parameters. Model-parallel implementations on the other hand have been successfully used on computer clusters, e.g. the COTS HPC, for models that have more than 11 billion parameters [20]. It requires roughly 82 GB of memory while most GPUs, e.g. the K80s used by AWS EC2, have only 12 GB available per node [10].

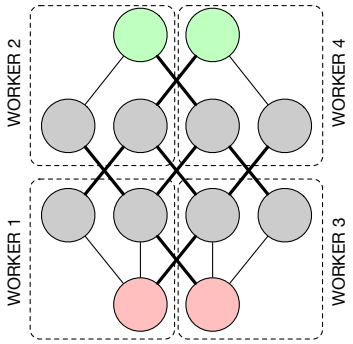


Figure 4.2: A single model can be deployed across multiple nodes, where every node uses the same training datasets, but computes the gradients for only a small portion of the model

4.2.2. Data Parallelism

Model parallelism enables further acceleration of the training phase, after the best suited hardware platform that is available has been selected. However, it works only with large-scale machine learning models where the gains in the computational time exceed the losses caused by communication overhead.

Another method of scaling the training phase, that can be used separately or together with model parallelism, is known as data parallelism. For data-parallel implementations, multiple copies of the same model are trained with unique batches of training data, as illustrated in Figure 4.3. Instead of dividing the parameters across multiple nodes that share the data as used by model parallelism, data parallelism divides the data across multiple nodes that share the parameters. Worker nodes contain model replicas and calculate the gradients for a given batch of data, which are then transferred to the parameter server for gradient averaging. However, the averaging of the gradients frequently results in decreased accuracy. The updated parameters are fetched by all of the worker nodes, which will then process the next batch of training data. It has to be noted, that data-parallel implementations can only be used if the entire model fits in the memory of a single node, otherwise it has to be used in conjunction with model parallelism.

Data parallelism can be either synchronous or asynchronous:

1. *Synchronous.* Gradients are averaged when all nodes have finished processing a chunk of data. Faster nodes have to wait for the slowest node in order to continue processing, therefore the training time of the entire model will be dependent on the execution time of the slowest node. However, the execution time can be very similar between nodes in certain settings, e.g. a single workstation with multiple GPUs.

2. *Asynchronous.* Parameter updates are performed independently of each other. It might lead to sub-optimal results, because model replicas might be trained on an outdated copy of the model parameters.

The scalability of data-parallel DNNs has been thoroughly evaluated by Keuper, et al. (2016) [36], who found that the widely-used data-parallel DNNs are becoming bound by the distribution overhead. Data-parallel implementations require the transfer of the weights and gradient updates for the entire model after every batch, which is the main cause of the overhead [36]. Secondly, training times for a single batch are low and will decrease even further when scaled to multiple nodes. The contrast between the compute and communication times is lower when a single multi-GPU node with fast communication buses is used, but the difference might reach multiple orders of magnitude on networks that use commodity Ethernet to copy data between nodes. Coates, et al. (2013) [20] have estimated the compute time for a gradient to be in the order of milliseconds per image, while it takes 8 seconds to copy the gradients or parameters between nodes when using map-reduce infrastructure with GPUs [24]. Paradoxically, using newer and faster hardware increases the overhead problem, because the training times for a single batch are decreased even further. This reduces the utilization of the compute units and increases the cost per training batch.

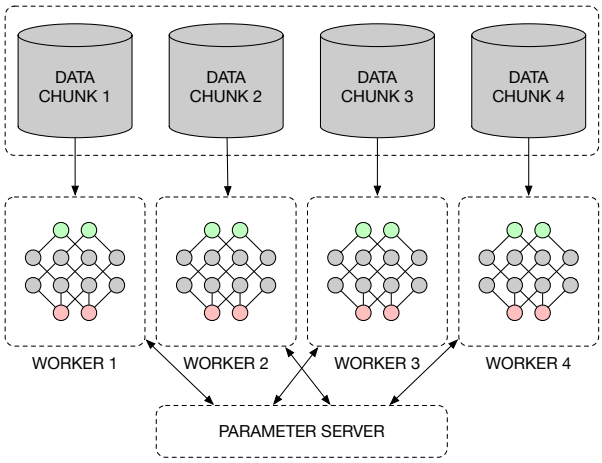


Figure 4.3: Data-parallel implementations compute the gradients for multiple copies of the same model with different datasets, which will then be averaged in the parameter server to get global gradient updates

Keuper, et al. state, that training a model becomes bound by the communication overhead already after 4 to 8 parallel nodes for the commonly used global batch size of 256. However, deep networks with less neurons per layer scale better than shallow networks with many neurons. Several approaches have been devised to overcome the network bandwidth bottleneck, e.g. eliminating unused weights, reducing the precision of the weights and compression of the transmitted weights. Nevertheless, these solutions do not address the fundamental issues caused by communication overhead. [36]

As communication overhead was observed to be a significant limitation of data parallelism, Keuper, et al. evaluated the speed-up on simulated nodes without any communication overhead. Global batch size B was kept constant, while the local batch size b was reduced to distribute the data equally between nodes. Linear speed-up could be expected if the communication overhead is removed. However, it was observed that the increases in the speed-up start to stall after distributing the data across a few nodes, i.e. as soon as the local batch size is drops below 32. After performing layer by layer analysis of the compute times, it was noticed that the only layer type that does not scale well on CPUs was the fully-connected layer. On GPUs, the speed-up for other layer types was reduced significantly. Since the fully-connected layers consume the largest portion of the execution time, the reduction in the speed-up of the other layers does not reduce the overall scalability initially. However, due to Amdahl's Law, they will eventually limit the scalability. An increase in the global batch size B has been proposed as a method to overcome the reduced scalability issues for $b < 32$. It has proven to scale almost linearly for up to 128 nodes, while $b > 32$. Larger batch sizes increase the computation time per iteration while reducing the number of iterations required until convergence, because more accurate gradients can be computed at every step. However, after reaching a certain global batch size B the computation time continues to increase while the number of iterations until convergence stays constant, which will limit further scalability. This can be overcome by increasing the step size in proportion to the increases in the global batch size, since larger batch sizes should provide more accurate gradient information and therefore enable larger steps without a decrease in the accuracy. Nevertheless, the experiments by Keuper, et al. indicate a significant decrease in the accuracy if this approach is taken. [36]

Data parallelism is widely-used by Google [50] and enables the scalability of models across hundreds of nodes. However, data-parallel implementations introduce a trade-off between the achieved speed-up and the degradations in the accuracy of the model. Therefore, the usage of data-parallel implementations is application specific depending on whether the achieved speed-up is worth the loss of accuracy.

4.2.3. Distributed Training

Finding the best hyperparameters for a single model can be tedious and time-consuming, since it is frequently performed by evaluating the performance of a range of hyperparameter combinations. As model-parallel and data-parallel implementations strive for the accelerated training of a single model,

it is required to train the same architectural models with different hyperparameter configurations sequentially. Another method of acceleration that could be used for this purpose is the parallelisation of the hyperparameter tuning across a computer cluster. It could be used together with model-parallel and data-parallel implementations, yet it becomes trivial to implement when used with modern cluster computing engines, e.g. Apache Spark. A number of copies of the same architectural model with different hyperparameters could be deployed across a cluster, while supplied with the same training datasets as illustrated in Figure 4.4. Furthermore, different network architectures could be evaluated in parallel, however the differences in the training times could force some nodes idle for extended periods.

Although this method works well for design space exploration, it is not a generic method of acceleration and will be of little use when the best network configuration has been found. However, this approach could be used to accelerate the training of a number of models that are used in combination for a single application. Many anomaly detection methods use a large number of input features to predict the value of a single variable, which is then compared to the measured value to detect anomalies, as described in Chapter 3. In a larger petrochemical system, the amount of predicted variables could increase to hundreds, therefore, the training of all of these models could be parallelised across a cluster, where each node computes the parameters for a model that predicts the value of a single variable.

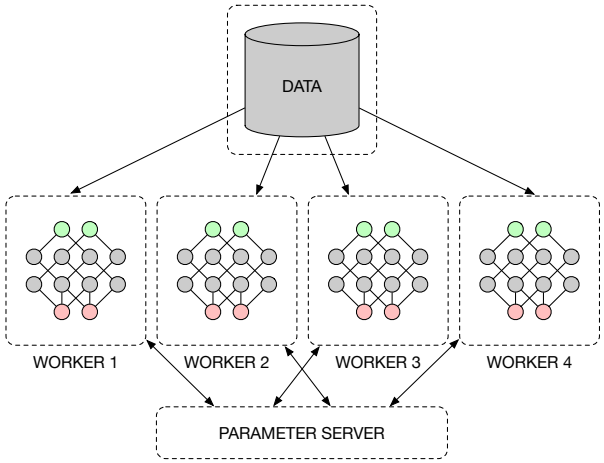


Figure 4.4: Multiple copies of a single model or multiple models with different hyperparameters can be trained across a computer cluster using the same training dataset

5

Implementation

This research project resulted in a proof of concept implementation for early fault prediction in Shell Pernis. It has been designed with the latest technologies and platforms available, while keeping it compatible with the platforms available at Shell. Section 5.1 gives an overview of the used technology stack as well as the goal and usage of each component. Section 5.2 provides the implementation details for RNNs, DNNs and ridge regression - the three methods used for machine learning. Finally, Section 5.3 provides sample implementations for the scalability of the models.

5.1. Technology Stack

One of the goals of this research project was to devise the machine learning approach in a way that enables Shell to easily integrate it with already existing technologies and infrastructure within the corporation. This limits the range of possible compute infrastructure that can be used for the project, however, several different options were still considered:

1. *Databricks.* Databricks is a serverless data analytics platform developed by the creators of Apache Spark, a big data processing engine that is built on top of the Amazon Web Services EC2 infrastructure.

2. *Google Cloud Platform.* Google Cloud Platform is one of the main competitors of Amazon Web Services, however, Amazon is the market leader by far in terms of market share.

3. *Servers at Shell.* Petrochemical companies, including Shell, utilize large computer clusters for seismic analysis that is used for the exploration of new hydrocarbon reserves. A variety of clusters are used, e.g. GPU clusters that have *** XYZ on *** nodes (anonymized due to confidentiality).

4. *Servers at Delft University of Technology.* The initial experiments were performed on the TU Delft servers, specifically a server node with K40 and GTX 750 Ti GPUs. However, this option is external to Shell and therefore could not be used for long-term application design.

Databricks was selected as the platform of choice, because the Google Cloud Platform was being decommissioned by Shell at the start of this research project; access to the clusters managed by Shell itself was limited; and access to corporate datasets outside of the company, e.g. Delft University of Technology, was limited. The resulting technology stack is depicted in Table 5.1, the components of which will be further discussed in this section.

Machine Learning Model
TensorFlow
Apache Spark
Databricks
Amazon Web Services EC2

Table 5.1: Technology stack

5.1.1. Amazon Web Services EC2

Amazon EC2 provides elastic compute clusters across a number of different instance types that are optimized for specific application domains. However, only a small portion of the instances are available via Databricks, many of which have already been replaced by newer AWS instances (C3 by C4, R3 by R4, and I2 by I3):

1. *Memory Optimized, R3.* R3 nodes are best for applications that are memory-intensive, e.g. databases. Optimized for low cost per used RAM.
2. *Compute Optimized, C3.* C3 provides high-performance nodes for applications that are CPU-bound. Optimized for low cost per computational performance.
3. *Storage Optimized, I2.* I2 nodes are optimized for applications that require low-latency access to significant amounts of permanent storage, e.g. large databases. Optimized for low cost per I/O operations.
4. *GPU Accelerated, P2.* P2 nodes are designed for compute-intensive applications that can easily be parallelized, e.g. machine learning, simulations.

AWS instances can be used by the hour with either on-demand or spot pricing. On-demand instances are more expensive, yet have guaranteed availability that is required for applications that cannot be interrupted. However, spot instances can be up to 90% cheaper than on-demand instances, as the spot instances are provided at a discount by AWS if the instances are underused. It may cause interruptions, but it is not an issue with batch jobs, such as training machine learning models. Furthermore, no differences between the availability of on-demand and spot instances was detected during this research project. Therefore, spot instances should be preferred when developing machine learning applications using Amazon EC2.

5.1.2. Databricks

Databricks is built around Apache Spark, an engine for large-scale data processing. Although Apache Spark has an integrated machine learning library, MLlib, it supports all of the major machine learning frameworks, e.g. TensorFlow, MXNet, Keras, Caffe, CNTK and Theano.

Databricks is a serverless platform, therefore, it is required to manage AWS EC2 clusters via Databricks' UI, as described in Section 5.1.1. Databricks can be connected to multiple external data sources, although Amazon S3 is the recommended service. Additionally, binary files can be uploaded to the Databricks' environment, which can then be accessed via active clusters. Source code can be developed and executed using data science notebooks via Databricks UI.

Clusters are composed of a single driver node and zero or more worker nodes. Driver nodes maintain the status of the notebooks attached to a cluster. Also, driver nodes control the cluster itself, as the Apache Spark master runs on the driver. Worker nodes on the other hand run Apache Spark executors that perform the actual computations. Databricks enables developers to run code on the driver without any workers, but this disables Apache Spark on the cluster. However, this disables access to the Databricks File System (DBFS). Therefore, at least two nodes have to be used in order to access files in DBFS. Databricks removes the complexity of cluster management by automating the creation,

management as well as configuration of the clusters. This is achieved by deploying prebuilt images that contain an installation of Ubuntu, Apache Spark, Apache Hadoop as well as the libraries for Java, Scala, Python, and R.

Databricks uses a distributed file system, Databricks File System, to manage data sources. It is built on top of Amazon S3, where a default Amazon S3 bucket is created to store the DBFS data when the account is created. Additionally, external Amazon S3 buckets can be mounted to DBFS. DBFS is capable of caching data from Amazon S3 to SSDs on worker nodes to speed up data access. It has to be noted, that every node has local storage that is separate from the global DBFS. Therefore, locally stored files have to be copied from *file:/* to *dbfs:/* before they can be accessed from the Databricks' UI. The file system can be controlled using similar commands to UNIX, as presented in Listing 5.1.

```
1 dbutils.fs.mkdirs("/tmp/")
2 dbutils.fs.head("/tmp/tmp.txt")
3 dbutils.fs.rm("/tmp/tmp.txt")
4 dbutils.fs.ls("file:/tmp/")
5 dbutils.fs.ls("dbfs:/tmp/")
6 dbutils.fs.cp("file:/tmp/src", "/tmp/dest")
```

Listing 5.1: Common commands for Databricks File System

In order to use the previously listed machine learning frameworks, it is required to install the frameworks on active clusters using initialization scripts. Listing 5.2 presents the initialization script used to install TensorFlow with CPU support on an active cluster, however, *tensorflow-gpu* has to be installed in order to utilize GPUs. However, it is not required to install TensorFlow and MXNet when using the latest version, as these frameworks will be pre-installed by default.

```
1 clusterName = ""
2
3 script = """
4 #!/usr/bin/env bash
5 set -ex
6 echo "***** Installing TensorFlow *****"
7 pip install tensorflow """
8
9 dbutils.fs.mkdirs("dbfs:/databricks/init/")
10 dbutils.fs.put("dbfs:/databricks/init/%s/install-tensorflow.sh" % clusterName, script, True)
```

Listing 5.2: TensorFlow initialization script

At the start of this research project it was possible to run TensorFlow on a single driver node without using Apache Spark. In the meantime, Databricks has increased its integration with TensorFlow by supporting TensorBoard, which is the visualization tool used to analyze applications using TensorFlow. A significant limitation in the integration with TensorFlow is the lack of support for distributed TensorFlow, however, Databricks has promised to support it in the near future.

5.1.3. Apache Spark

Apache Spark is based on the famous MapReduce programming model that is used by several big data engines, e.g. Apache Hadoop. It makes the framework scalable and flexible, as the number of computational nodes as well as the structure of the cluster can easily be changed. Furthermore, Apache Spark is fault-tolerant as the state can always be restored without any data loss when a node in the cluster fails. Apache Hadoop, the most popular large-scale computing framework before the introduction of Apache Spark, writes the data to disk after every operation. Disk access is significantly slower than memory access, therefore, Apache Spark uses in-memory computing to speed up the data access. As a result, Apache Spark can reduce the execution time of memory-intensive applications by more than 100 times when compared to Apache Hadoop [18].

Apache Spark is built on the resilient distributed dataset (RDD) abstraction, which is a set of objects distributed across a computer cluster. RDDs can be created by either reading an external dataset or by parallelizing a collection. The operations that can be performed on RDDs are transformations, that modify an already existing RDD, and actions, that perform a computation on workers in order to obtain a value on the driver. The operations provided by Apache Spark are lazy in the sense, that the

operations will only be carried out when an action is performed on an RDD.

Apache Spark is not required for performing inference, however, it enables scalable transformations on the input dataset in order to convert it to the format required for the training phase of machine learning. The input datasets can be read from DBFS using Apache Spark commands as shown in Listing 5.3. The RDD created from the external data source can then be modified using a number of common MapReduce transformations such as *map(·), filter(·), join(·)* and actions such as *reduce(·), collect(·)*. Finally, the new RDD can be written to DBFS, as shown in Listing 5.4.

```
1 csv_file = sc.textFile('/FileStore/tables/__hash__/file.csv')
2 reader = csv.reader(csv_file.collect())
3
4 for row in reader:
5   ...
6   ...
```

Listing 5.3: Reading files from DBFS

```
1 ...
2 ...
3
4 rdd.saveAsTextFile("users/username/file.csv")
```

Listing 5.4: Writing RDDs to DBFS

5.1.4. TensorFlow

TensorFlow is an open-source machine learning library developed by Google. The initial version was released to the public in Q4 of 2015 and it has become the most popular open-source machine learning library in the meantime. Furthermore, its popularity has increased due to its wide usage in massive open online courses (MOOCs). TensorFlow provides high-level APIs, i.e. *tf.contrib.learn*, for fast and simple implementations as well as low-level APIs, i.e. TensorFlow Core, for developers that require more control over their applications. Furthermore, TensorFlow Lite, a modified version of TensorFlow that is targeted for Android, was announced in Q2 of 2017.

TensorFlow is based on the concept of computational graphs, which are composed of a series of operations that define the flow of the program. However, the operations are only executed when the graph is explicitly run. TensorFlow uses the concept of sessions to run graphs, where sessions manage the state and the control of the applications. Therefore, the applications built using TensorFlow are constructed of two separate steps. TensorFlow boasts more than a hundred operations, that range from the creation of constants and math operations to ANN-specific operations. The inputs and outputs of these operations are tensors, which are the basic data structures in TensorFlow. Tensors are arrays that can be of arbitrary dimensions, which is known as the rank of a tensor. Graphs can be created with static data, however, it is almost always required to pass data into the graph, e.g. training and testing data. External data can be injected to the graph using placeholders, which are entry points to a graph that require data to be passed via these entry points to the graph during graph execution. TensorFlow stores model parameters in variables, that can be initialized to a predefined value or distribution and will then be iteratively modified by the optimizers in order to reduce the cost function.

Listing 5.5 presents a typical TensorFlow application for the training phase. The training dataset is separated into batches, as the entire dataset does not fit into the memory. TensorFlow graphs are executed using *sess.run(·)*, which can also be used to fetch the values of the graph nodes, e.g. the cost. The number of training epochs can be static, however, early stopping is frequently used to train the model until the cost stops decreasing.

```
1 def train(self, dataset):
2   start_time = time.time()
3
4   with tf.Session() as sess:
5     # Initialize variables tf.Variable
6     sess.run(tf.global_variables_initializer())
```

```
7
8     # Train the model
9     for i in range(Config.train['epochs']):
10      # Shuffle the dataset between epochs for better training
11      random.shuffle(dataset)
12      cost = []
13
14      for j in range(len(dataset) / Config.train['batch_size']):
15        features, target = [], []
16        for k in range(Config.train['batch_size']):
17          # Extract the inputs and outputs from the dataset
18          features.append(dataset[Config.train['batch_size'] * j + k][0])
19          target.append(dataset[Config.train['batch_size'] * j + k][1])
20
21        _, cost_batch = sess.run([self._optimizer, self._cost],
22              {self._input: features, self._output: target, self._keep_prob: Config.rnn['
    dropout']})
23        cost.append(cost_batch)
24
25      print "Epoch: ", i + 1, "; Cost: ", np.mean(np.array(cost))
26
27    # Save the trained model for later usage
28    saver = tf.train.Saver()
29    saver.save(sess, Config.path + 'logs/model')
30
31  print "Training Time:", (time.time() - start_time)
```

Listing 5.5: TensorFlow batch training

5.2. Machine Learning

5.2.1. Pipeline

The machine learning pipeline that should be used when deploying the current research in Shell Pernis is depicted in Figure 5.1. The data is stored in OSISoft PI, which can be accessed via an OSISoft PI server. It enables access to historical as well as live data, since both are required for making early fault predictions. The machine learning models are interchangeable within a range of approaches, e.g. ridge regression, DNNs and RNNs, as long as the inputs and the outputs of the model are the same. The inputs to the model have to be cleaned, therefore, the time series that contain controllers that are in manual mode have to be discarded. Otherwise, the relationships between the *OPs*, *SPs* and the *PVs* might not be correct, as the operators could have fixed some of the values while other values would still change based on the actual state of the system. Furthermore, the normal data selection method described in Section 5.2.4 should be performed before the actual training of the machine learning model.

Model training is performed on high-performance nodes in order to reduce the training time. However, the real-time predictions should be performed on low-performance nodes that keep the costs of the early fault prediction applications low. Then, the difference between the predicted values and the actual values can be used to detect anomalies in the live data. This can be extended by various methods, e.g. the period during which the average error has been rising or the time by when the monitored parameter will increase to the maximal value. Finally, a notification has to be sent to the operators by integrating the developed solution for early fault prediction with the user interfaces used by the operators of the refinery.

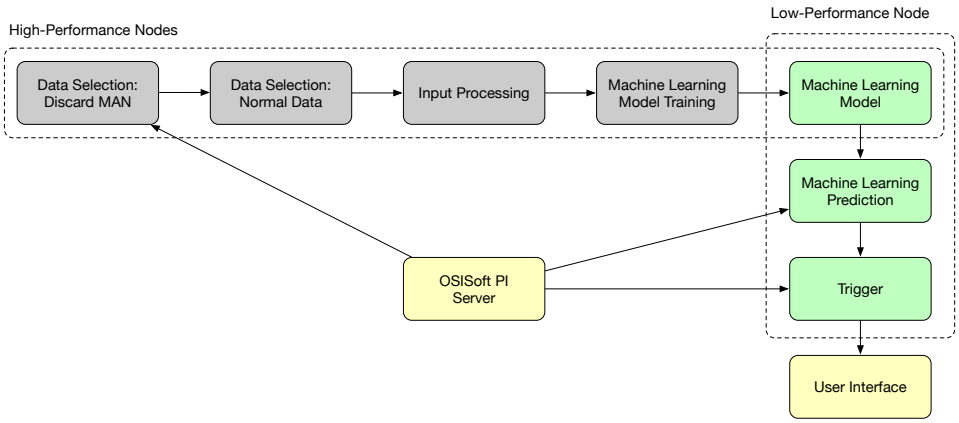


Figure 5.1: Machine learning pipeline for the pilot

The machine learning pipeline used to evaluate the methods during this research, however, lacks the integration with Shell-specific application. First of all, the OSISoft PI server was not available and the historical data was provided as a CSV file. However, this made it impossible to evaluate the proposed solutions using live data. Secondly, the solution was not integrated with the user interfaces in the refinery, instead, the outputs were mapped to PDFs in order to evaluate the performance of methods. Finally, the training and prediction phase was performed on a single node instead of a separate high-performance node for training and a low-performance node for prediction in order to speed up development process.

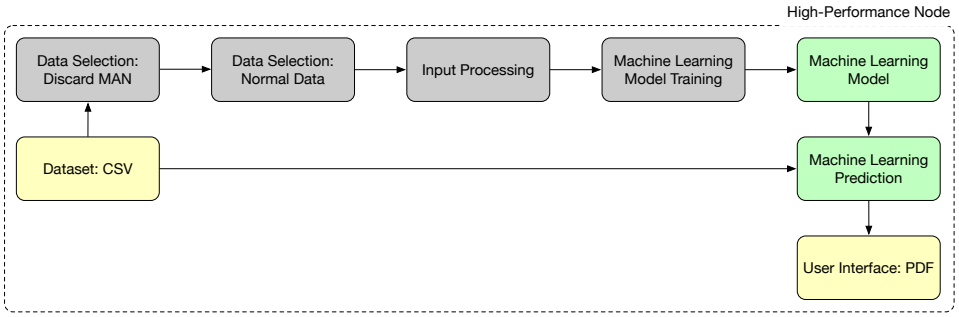


Figure 5.2: Machine learning pipeline for the PoC

5.2.2. Recurrent Neural Networks

Many to one RNNs are used in this research project, which means that a scalar value is predicted based on a sequence of prior inputs across a number of features. Graph definition has to be started by creating placeholders for feeding data to the graph when executing the graph. RNNs require placeholders for the training batch, the respective predictions as well as the probability for the dropout layer, as shown in Listing 5.6. The input to the model has to be 3-dimensional and the output 2-dimensional, as the training will be performed in batches and for every entry in the batch there will be a sequence of features that is used to predict a single scalar value. Dropout requires a probability lower than 1 in order to perform regularization on the network when training, however, the probability has to be 1 in order to make use of the entire network during inference. Therefore, the probability has to be fed to the graph separately for training and inference.

```
1 # Input Format: [Batch of Sequences, Sequence Length, Features]
2 self._input = tf.placeholder(tf.float32, [Config.train['batch_size'], Config.rnn['length'],
      Config.input['features']])
3 # Output Format: [Batch of Sequences, Predicted Value]
4 self._output = tf.placeholder(tf.float32, [Config.train['batch_size'], 1])
5 # Dropout Probability
6 self._keep_prob = tf.placeholder(tf.float32)
```

Listing 5.6: Feeding data to the RNN graph

TensorFlow supports multiple RNN cell types, including the vanilla RNN cell, GRU, basic LSTM cell and the more complex LSTM cell. GRUs and basic LSTMs were evaluated during this research project to see, whether one is superior to the other for this problem domain. RNN cells can be configured to use different activation functions, as described in Section 5.2. Furthermore, the number of units in an RNN cell, one of the most important RNN parameters, has to be selected when defining the graph. It represents the learning capacity of the RNNs, as it defines the width of the vectors that are passed between the structures of the RNN cell, as described in Section 3.5.3. Finally, dropout has been used to in between the layers in order to avoid overfitting, which can easily happen when overdimensioning the network.

```
def rnn_cell():
  # Create the RNN cell
  if Config.rnn['cell'] == 'LSTM':
    rnn = tf.contrib.rnn.BasicLSTMCell(Config.rnn['units'],
          activation = Config.rnn['activation'], state_is_tuple = True, reuse = tf.
    get_variable_scope().reuse)
  elif Config.rnn['cell'] == 'GRU':
    rnn = tf.contrib.rnn.GRUCell(Config.rnn['units'],
          activation = Config.rnn['activation'])

  # Apply dropout
  return tf.contrib.rnn.DropoutWrapper(cell = rnn, output_keep_prob = self._keep_prob)
```

Listing 5.7: RNN cell definitions

More complex time series relations can be modelled by stacking multiple RNN layers on top of each other, as shown in Listing 5.8.

```
# Stack multiple RNN cells on top of each other
rnn = tf.contrib.rnn.MultiRNNCell([lstm() for _ in range(Config.rnn['layers'])],
    state_is_tuple = True)
```

Listing 5.8: Stacked RNN cells

The RNN is created using the *dynamic_rnn(·)* function, which dynamically unrolls the network. It uses a loop in order to dynamically build the graph during execution. The dimensions of the input batches can be of variable size, as the graph is created during runtime. Historically, it was possible to create fixed length RNNs using the slightly slower *rnn(·)* function, however, it has been removed in the latest versions. It further illustrates the fast changes in the TensorFlow framework, as new functionality is constantly introduced and outdated functionality is phased out. Therefore, TensorFlow is an attractive framework that is always up to date, however, it makes it harder for companies to maintain a stable codebase as prior implementations might not work with subsequent revisions.

The RNN implementation provides the output of the cell for every time step in the unrolled RNN, however, only the last output is required in this research project as many to one RNNs have been used. Therefore, the 3-dimensional RNN output has to be transformed to a 2-dimensional representation, which only has the outputs of the last unrolled RNN cell. This can be achieved by transposing the raw RNN output and gathering only the desired output, as shown in Listing 5.9.

```
# Unroll the RNN dynamically
# Ignore state, since we use stateless RNNs
rnn_output, _ = tf.nn.dynamic_rnn(rnn, self._input, dtype = tf.float32)

# Input Format:  [Batch of Sequences, Sequence Length, RNN Units]
# Output Format: [Sequence Length, Batch of Sequences, RNN Units]
rnn_output = tf.transpose(rnn_output, [1, 0, 2])

# Get the output of the last unrolled RNN cell
# Output Format: [Batch of Sequences, RNN Units]
rnn_output = tf.gather(rnn_output, int(rnn_output.get_shape()[0]) - 1)
```

Listing 5.9: RNN definition for many to one implementation

The final part of the RNN graph definition is the cost function and optimization. A variety of operations are provided by TensorFlow that reduce the dimensions of a tensor, e.g. the sum, mean, max and min operations. However, the sum of the squared errors has been used as the target for the optimizer.

Exponential decay has been used to gradually decrease the learning rate, where the learning rate is decreased after every epoch. Multiple optimizers were evaluated, including *GradientDescentOptimizer*, however, the differences in the predictions were small. Eventually, *AdamOptimizer* was selected, as described in Section 5.2.

```
# Define the loss function
self._cost = tf.reduce_sum(tf.pow(regressor - self._output, 2))

# Reduce the learning rate after every epoch
# decayed_learning_rate = learning_rate * decay_rate ^ (global_step / decay_steps)
global_step = tf.Variable(0, trainable = False)
decay_rate = tf.train.exponential_decay(
    Config.optimizer['learning_rate'],
    global_step,
    Config.optimizer['decay_steps'],
    Config.optimizer['decay_rate'],
    staircase = True    # Decay at discrete intervals (after every epoch)
)

# Define the optimizer
optimizer = tf.train.AdamOptimizer(decay_rate)
self._optimizer = optimizer.minimize(self._cost, global_step = global_step)
```

Listing 5.10: Cost function and optimizer for RNNs

5.2.3. Deep Feedforward Neural Networks

Changing the machine learning model from RNNs to DNNs is straightforward, as the structure of the implementation is very similar. The fundamental difference between RNNs and DNNs is the usage of sequences instead of scalar features. Therefore, the placeholder for the input feed has to be modified from a 3-dimensional structure to a 2-dimensional structure.

DNN implementations use variables for parameters instead of cell structures as used for RNNs. Listing 5.11 features a two-layer ANN, as this structure achieved the lowest MSE for the defined problem. The weights have been initialized to small numbers using the normal distribution and the biases have been initialized to zero, as suggested in [7]. Then, two layers of *tanh* activation functions with interleaved dropout have been used. Finally, the outputs of the neural layers are combined to a single neuron that can be used to predict a scalar value.

```
# Input Format: [Batch of Sequences, Features]
self._input = tf.placeholder(tf.float32, [Config.train['batch_size'], Config.input['features']])
# Output Format: [Batch of Sequences, Predicted Value]
self._output = tf.placeholder(tf.float32, [Config.train['batch_size'], 1])
# Dropout Probability
self._keep_prob = tf.placeholder(tf.float32)

# Definition of weights and biases
W1 = tf.Variable(tf.truncated_normal([Config.input['features'], 72], stddev = 0.3))
W2 = tf.Variable(tf.truncated_normal([72, 72], stddev = 0.3))
W3 = tf.Variable(tf.truncated_normal([72, 1], stddev = 0.3))
B1 = tf.Variable(tf.zeros([72]))
B2 = tf.Variable(tf.zeros([72]))
B3 = tf.Variable(tf.zeros([1]))

# Non-linear activations
Z1 = tf.tanh(tf.add(tf.matmul(self._input, W1), B1))
Z1 = tf.nn.dropout(Z1, self._keep_prob)
Z2 = tf.tanh(tf.add(tf.matmul(Z1, W2), B2))
Z2 = tf.nn.dropout(Z2, self._keep_prob)

output = tf.add(tf.matmul(Z2, W3), B3)
```

Listing 5.11: Changes required for DNNs when compared to RNNs

5.2.4. Ridge Regression

Ridge regression is used as the baseline model for this research project, as it was found to be the best linear regression approach for petrochemical refineries by Bender, et al (2016) [23]. However, ridge regression will also be evaluated for automatic selection of training data. The implemented anomaly detection method creates a model for normal operating conditions of the petrochemical refinery. Therefore, selection of the training data is of utmost importance as large neural networks can easily fit to abnormal conditions even if they persist only for a few months out of years of training data. Theoretically, such data selection could be performed manually or by using application-specific rules. However, it would require a small army of petrochemical refinery experts to select the timeframes this way, as the dataset might include thousands of sensors from over a period of more than ten years. A subset of the abnormal time series could be detected using the maintenance logs stored in SAP, however, some petrochemical systems might behave abnormally even if they do not cause an issue that is logged into the maintenance system. Furthermore, the goal of this research project was to devise a scalable anomaly detection method that requires minimal to no inputs from industry experts.

Normal data could be selected by ridge regression when using the training set for predictions on the already trained model. Then, a certain percentile of the training examples with the largest prediction error or the training example with an error that is multiple times larger than the median could be used to remove abnormal data points. Furthermore, certain time series could be excluded based on the density of training examples that are labelled as abnormal. Ridge Regression might be suitable for this as it is a statistical method, although it is considered to be part of machine learning.

Similarly to DNNs, the implementation of ridge regression using TensorFlow requires the usage of variables for the storage of weights and biases. However, ridge regression is based on a single matrix multiplication instead of multiple layers of non-linear activation functions, as was used for DNNs. The key difference for the ridge regression implementation is the addition of the L_2 regularization to the cost function, as shown in Listing 5.12. It can easily be calculated by the *global_norm(·)* function provided by TensorFlow, the output of which can then be multiplied by regularization parameter λ.

```
# Definition of weights and biases
W1 = tf.Variable(tf.truncated_normal([Config.input['features'], 1], stddev = 0.3))
B1 = tf.Variable(tf.zeros([1]))

# Matrix multiplication
output = tf.add(tf.matmul(self._input, W1), B1)

# Define the cost function
cost = tf.reduce_mean(tf.pow(output - self._output, 2))
# Apply ridge regression
self._cost = tf.add(cost, tf.multiply([Config.regression['lambda'], tf.global_norm([W1])))
```

Listing 5.12: Changes required for ridge regression when compared to DNNs

5.3. Scalability

5.3.1. Model Parallelism

By default, TensorFlow deploys the graph on the node with the lowest ID. However, the distribution of the graph nodes between multiple GPUs or nodes can be explicitly defined using *with tf.device(·)*. This enables the distribution of certain parts of a model across a number of GPUs. TensorFlow automatically handles the connections between the different nodes, therefore, only small changes are required in order to distribute a model that was designed for a single node between multiple nodes. However, the model has to be fairly large in order to achieve superior performance when compared to training on a single node. An example of model parallelism for ANNs is presented in Listing 5.13.

```
...
results = []

# Distribute the ANN between multiple GPUs
```

```
for d in ['/gpu:0', '/gpu:1']:
  with tf.device(d):
      self._input.append(tf.placeholder(tf.float32, [Config.train['batch_size'], Config.rnn['length'], Config.input['features']]))

      input = self._input[len(self._input) - 1]

      W1 = tf.Variable(tf.truncated_normal([Config.input['features'], 64], stddev = 0.3))
      B1 = tf.Variable(tf.zeros([64]))
      Y1 = tf.tanh(tf.add(tf.matmul(input, W1), B1))
      Y1 = tf.nn.dropout(Y1, self._keep_prob)
      W2 = tf.Variable(tf.truncated_normal([64, 64], stddev = 0.3))
      B2 = tf.Variable(tf.zeros([64]))
      Y2 = tf.tanh(tf.add(tf.matmul(Y1, W2), B2))
      Y2 = tf.nn.dropout(Y2, self._keep_prob)
      results.append(Y2)

# Combine the results of multiple GPUs
with tf.device('/cpu:0'):
  W5 = tf.Variable(tf.truncated_normal([64, 1], stddev = 0.3))
  W6 = tf.Variable(tf.truncated_normal([64, 1], stddev = 0.3))
  B6 = tf.Variable(tf.zeros([1]))
  self._regressor = tf.matmul(results[0], W5) +
                    tf.matmul(results[1], W6) + B6

  # Define the loss function
  self._output = (tf.placeholder(tf.float32, [Config.train['batch_size'], 1]))
  self._cost = tf.reduce_mean(tf.pow(self._regressor - self._output, 2))

...
```

Listing 5.13: An example of model parallelism

5.3.2. Data Parallelism

Data-parallel implementations store the parameters on a single node while calculating the gradient updates on multiple nodes. TensorFlow enables the storage of variables on specific nodes using *with tf.device(·)*, as illustrated in Listing 5.14.

```
# Store the parameters on a specific device
with tf.device('/cpu:0'):
  __parameter__ = tf.get_variable(__name__, __shape__, initializer=__initializer__, dtype=tf.float32)
```

Listing 5.14: Parameter storage on a specific device

The placeholders and the graphs have to be created on multiple nodes similarly to model-parallel implementations. However, the gradients are computed on each node but not applied to the graph. Instead, references to the gradients are stored on each node and the average of those gradients is applied to the graph on a single node when all nodes have finished computing the gradients. Therefore, the implementation is synchronous as the gradients are only applied when the slowest node has finished computing.

```
...
# References of the distributed gradients
gradients = []

# Create the ANN graph across multiple GPUs
for d in ['/gpu:0', '/gpu:1']:
  with tf.device(d):
    # References of the distributed placeholders
    self._input.append(tf.placeholder(tf.float32, [Config.train['batch_size'] / Config.train['gpus'], Config.rnn['length'], Config.input['features']]))
    self._output.append(tf.placeholder(tf.float32, [Config.train['batch_size'] / Config.train['gpus'], 1]))
    ...
    # Compute the gradient for a single GPU
    g = optimizer.compute_gradients(self._cost)
```

```
14      gradients.append(g)
15
16  with tf.device('/cpu:0'):
17    # Apply the average of the gradients gradient
18    self._gradients = optimizer.apply_gradients(self.average_gradients(gradients))
19
20  ...
```

Listing 5.15: Data-parallel graph creation

The dataset has been split between the nodes for data-parallel implementations, therefore, the inputs have to be injected to the graph via multiple placeholders as shown in Listing 5.16.

```
1  ...
2  # Inject multiple batches to the distributed placeholders
3  _ = sess.run(self._train,
4               {self._input[0]: features[0], self._output[0]: targets[0],
5                self._input[1]: features[1], self._output[1]: targets[1]})
6  ...
```

Listing 5.16: Data-parallel graph execution

6

Experimental Results

This chapter presents the main results obtained during the research project. Section 6.1 discusses the design space exploration for the RNNs that was performed in order to find the optimal hyperparameters. Section 6.2 and Section 6.3 present the results for the two use cases at Shell Pernis that were evaluated during this research project. Finally, Section 6.4 evaluates the suitability of ridge regression for the selection of the normal operating data and Section 6.5 presents the speed-up obtained by utilizing different scalability methods.

6.1. Design Space Exploration

The main focus of this research project was on the implementation and evaluation of RNNs for the prediction of control valves failures, the configuration of which has a significant effect on the difference between the prediction and the actual values. The model is trained with the features listed in Section 2.1, while the output consists of a single real-valued prediction. The evaluation was started by tuning the hyperparameters of the RNNs with the objective of minimizing the prediction error for the normally behaving control valves based on use case 1. An important evaluation metric is the mean squared error (MSE), which represents the quality of an estimator based on the average of the squares of the errors. The tuning was started by plotting the MSE for a 4-layer RNN with a range of RNN unit sizes, as this hyperparameter has the most significant impact on the prediction quality. The MSE decreases significantly until the RNN size reaches 512 as illustrated in Figure 6.1. However, RNNs with a size of 512 introduce limitations for the maximal historical length as well as the number of RNN layers, because the model does not fit into the memory of a single GPU when using a multi-layer RNN with 512 units and a historical length longer than 10. Therefore, the number of RNN units was limited to 256, which provides sufficient possibilities for the historical length while keeping the MSE relatively low.

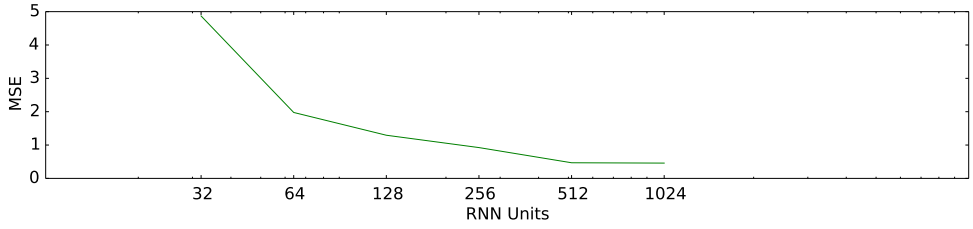


Figure 6.1: MSE for a range of RNN units

Similarly, the MSE of a 256 unit RNN for a number of RNN layers was evaluated, as plotted in Figure 6.2. Lowest MSE can be achieved with a 5-layer RNN, however, 4-layer RNNs were used for the evaluation of RNNs later on as it enabled the usage of longer historical lengths. A significant difference between Figure 6.1 and Figure 6.2 is the tendency of the error rate to decrease for the first additional

RNN layers and to increase when a larger number of RNN layers is used. It means, that the model becomes too large for the amount of data and it starts overfitting to the noise in the training set.

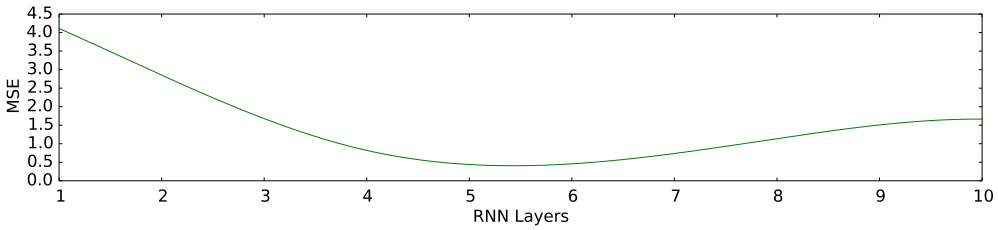


Figure 6.2: MSE for a range of RNN layers

RNNs were chosen as the preferred machine learning method primarily due to its capability to learn from sequences instead of discrete values. The MSE for different sequence lengths is presented in Figure 6.3, which is therefore important for the evaluation of the suitability of RNNs for the use cases at Shell Pernis. It was presumed prior to the evaluation that RNNs could reduce the MSE for the normally operating control valves several times by using sequence lengths longer than one. However, increasing the sequence length from one to four reduced the MSE only by 40%. Furthermore, the error rate quickly rises when increasing the sequence length to more than 7 samples (i.e. 7 minutes). This might be caused by several issues, as described in Chapter 7.

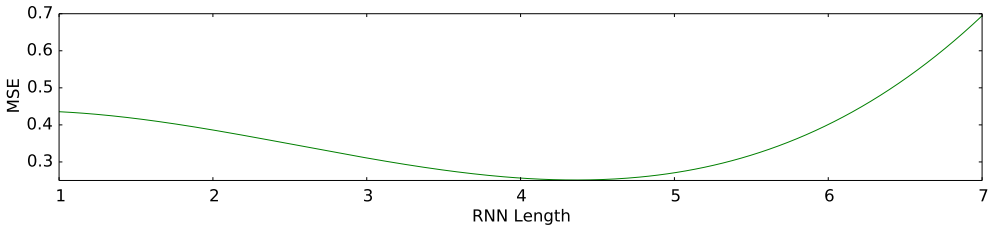


Figure 6.3: MSE for a range of RNN sequence lengths

The design space exploration for the number of RNN units and layers was performed for LSTMs with *tanh* activation function. However, $ReLU$ activation functions and GRU cell types are commonly used. Therefore, the training and testing MSE was evaluated for all combinations as presented in Figure 6.4. The hyperparameters for the new activation functions and unit types were based on the best configuration achieved during the prior DSE for an LSTM with *tanh* activation function - a 4-layer RNN with 256 units and a historical length of 4. However, the evaluation of the new combinations was performed with slightly modified hyperparameters, especially the learning rate, as different activation functions yield the lowest MSE with different hyperparameters.

It is clearly visible that the *tanh* activation function performs better than the $ReLU$ activation function for both GRUs and LSTMs. The poor performance of the $ReLU$ activation function is presumably caused by neurons that never fire, which is caused by too large gradients. The difference between LSTMs and GRUs while using the *tanh* activation function is relatively low, as LSTMs outperform GRUs by less than 5% for the training set. However, GRUs outperform LSTMs by more than 15% when using the testing set. Therefore, the more complex structure of the LSTMs probably causes overfitting to the training set which degrades the results for the testing set. GRUs on the other hand provide similar performance on the training set while increasing the performance on testing set, therefore, GRUs generalize better than LSTMs for this problem.

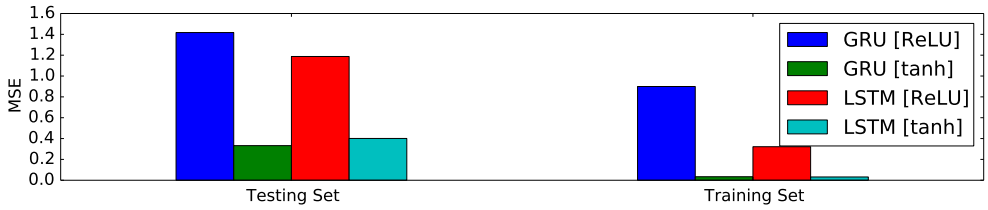


Figure 6.4: MSE for the most common RNN types and activation functions, where lower MSE values represent better solutions

The simplified structure of GRUs when compared to LSTMs should result in lower execution times when using GRUs. The experiments performed during this research project verified this presumption, as presented in Figure 6.5. The difference between the execution times for GRUs with *ReLU* and *tanh* activation functions is less than 1%. However, LSTMs with *tanh* activation functions take more than 7% longer to train than GRUs with *tanh* activation functions.

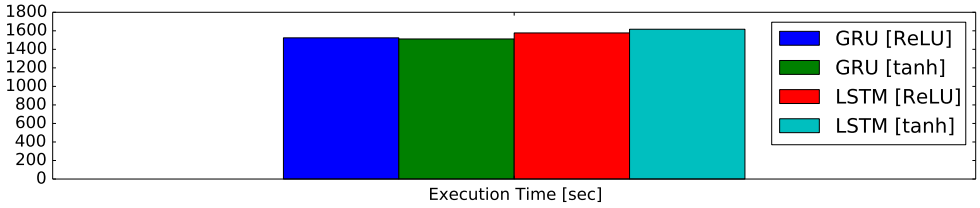


Figure 6.5: Execution times for the most common RNN types and activation functions in seconds

GRUs with the *tanh* activation function provided the best results in terms of the MSE as well as the lowest execution times, therefore, it was chosen as the preferred method for RNNs.

6.2. Use Case 1

Experiments for the use case 1 were performed on multiple control valves. The target was to verify, whether the machine learning methods are capable of distinguishing normally behaving valves from abnormally behaving valves. Therefore, the difference between the prediction and the measured values, i.e. the error rate, should be as low as possible for the normal valves while being as large as possible for the abnormal valves. The size of the training and testing set used during the evaluation was limited, although the data was available for the past 10 years due to organizational restrictions. Therefore, the training phase was performed on a single year's worth of training data and a single year's worth of testing data. The implemented approach models the normal behaviour of the control valves, therefore, the dataset had to be cleaned prior to the training phase. Firstly, it was achieved by selecting the training data together with the personnel at Shell Pernis. Secondly, the entire dataset was cleaned from time series where controllers were set to manual mode.

The results were concatenated into a single graph, in order to provide an overview of the trends in the data. The prediction as well as the measured value for the abnormally behaving control valve is plotted in Figure 6.6 and the resulting difference between the two graphs in Figure 6.7. It is clearly visible that the average difference between the prediction and the measured value increases over a period of 3 months, after which there is a large spike in the difference. This was caused by the shutdown of the crude distiller XYZ in order to repair the abnormally behaving valve. However, the error rate remains high even after the replacement of the broken valve for the last 2 months of the graph. Furthermore, the difference between the two graphs increases significantly for a few weeks, after which the difference between the two stabilizes on a relatively constant average value.

Firstly, the high error rate after the replacement of the valve is presumably caused by the different behaviour of the new valve when compared to the old valve in its normal operating range. It remained

unanswered, whether this was caused by the lack of training data as the previous control valve was installed before the start of the used training data or whether the model has to be retrained after the replacement of control valves. Therefore, it should be further evaluated whether increasing the training set from one year to ten years makes it possible to reduce the prediction error after the replacement of the valve or whether the model has to be retrained after the replacement of the valves. Secondly, the higher error rate after the replacement of the broken control valve during the course of a few weeks was presumably caused by an issue in another component or by the different behaviour of the crude distiller altogether, because a similar increase in the error rate was also present in the normal valves as presented in Figure 6.9.

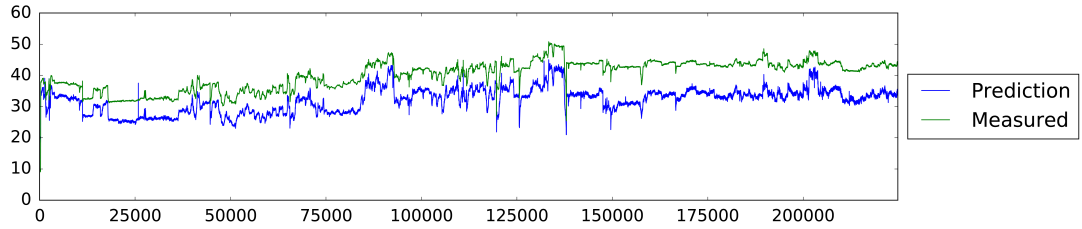


Figure 6.6: Actual and predicted values for an abnormal valve over a period of 5 months

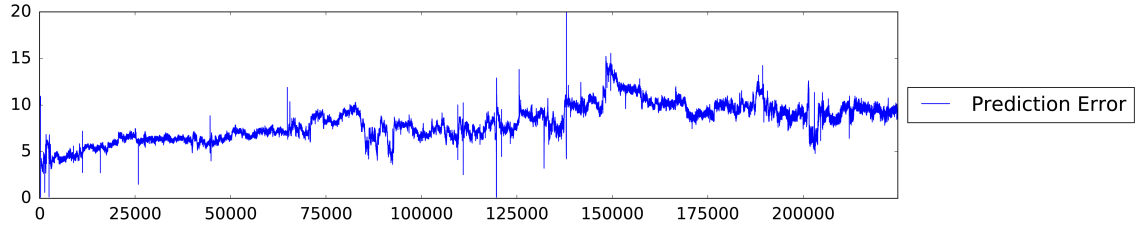


Figure 6.7: Difference between the actual and predicted values for an abnormal valve over a period of 5 months

The error rate for the normally operating control valves should be as low as possible. The prediction as well as the measured value for one of the normal control valves is plotted in Figure 6.8 and the resulting difference between the two graphs in Figure 6.9. The resulting graphs were similar for other normally operating valves, although a single plot is provided in this report for brevity. The error rate is significantly lower than the error rate for the abnormal valves, especially, when compared to the predicted and measured values as they are twice as high for the normally operating valve when compared to the abnormally operating valve. However, there are more than 10 spikes in the error rate during the 5-month period, which would invoke a notification when a simple comparison is used for the trigger. Although some spikes occur directly before one of the other controllers is set to manual mode, more than half of the spikes still take place without any imminent issues in the monitored subsystem. Therefore, it might be required to set a minimum time during which the error rate has to be higher than a defined threshold in order to avoid false positives, as it decreases the trust of the operators in the developed solution.

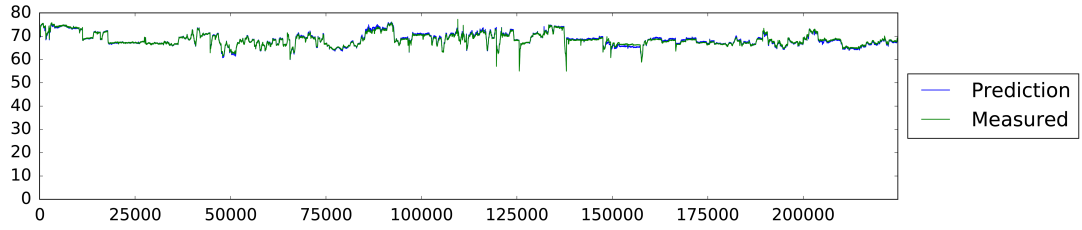


Figure 6.8: Actual and predicted values for a normal valve over a period of 5 months

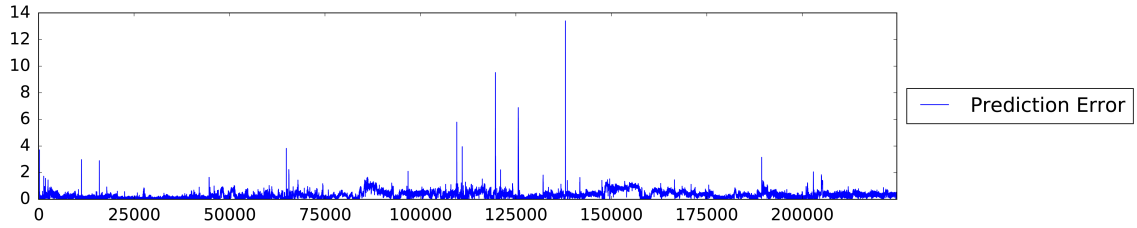


Figure 6.9: Difference between the actual and predicted values for a normal valve over a period of 5 months

The different machine learning methods that were evaluated resulted in similar prediction graphs, however, there were still slight differences when compared using MSE and median. MSE provides good evaluation criteria for normal conditions, however, it does not always provide the best metrics for abnormal conditions as smaller sequences of higher error rates can easily increase the overall MSE value. Therefore, the median error rate was used to compare the different methods, as it provides better representation of the overall trends. Ridge regression and RNNs achieved similar results for the normal operating conditions while DNNs had more than twice as high median error rate as illustrated in Figure 6.10. However, RNNs were able to provide higher increases for median error rates when predicting the values of the abnormally operating valve. Therefore, the ratio between the median error rates for abnormally operating valves and the normally operating valves is the highest for RNNs as presented in Figure 6.11.

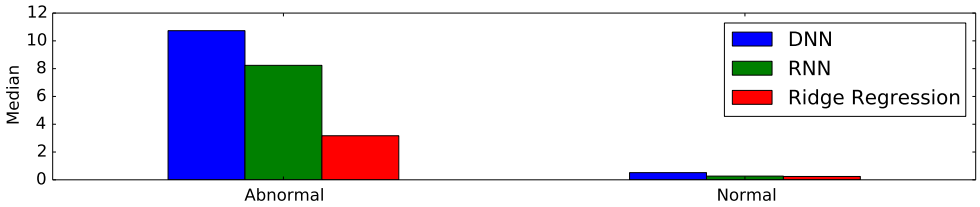


Figure 6.10: Median error rate for the evaluated machine learning methods, where solutions with high median error rate for the abnormally behaving control valve and low median error rate for the normally behaving control valve represent better solutions

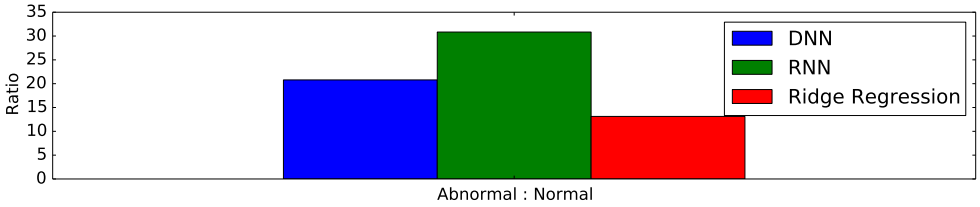


Figure 6.11: Ratio between the median error rates for abnormally and normally behaving control valves, where higher ratios represent better solutions

Another important criterion when comparing the machine learning methods is its scalability in terms of the number of input features. The prior graphs were plotted for the entire dataset, i.e. 148 features, yet it is desirable to compare it with a smaller section if the furnace with 48 features that represent only the parallel control valves of the furnace. Figure 6.12 and Figure 6.13 present the MSE for both ridge regression and RNNs with 48 features and 148 features, as they provided superior results to DNNs. The MSE for the normally operating valves increases with the number of features, which is inevitable as the complexity of the modelled system as well as the noise increases. However, the MSE for the abnormally operating valve increases with the number of features when using RNNs while it decreases when using ridge regression. Therefore, it could be assumed that RNNs provide better anomaly detection capabilities when using high-dimensional features and ridge regression yields better results for low-dimensional features.

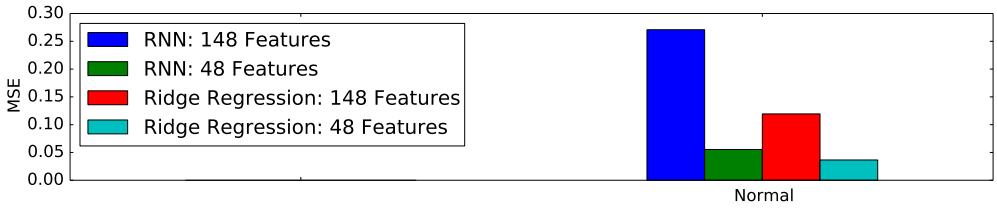


Figure 6.12: MSE for normally operating valves, where lower MSE values represent better solutions



Figure 6.13: MSE for abnormally operating valves, where higher MSE values represent better solutions

A fundamental factor that enabled ridge regression to obtain similar results to RNNs in this specific use case was that the control valves behaved linearly during the evaluation period. However, control valves behave non-linearly in their entire operating range. As the used ridge regression approach uses statistics to obtain a line that best represents a linear relationship, it is natural that it reaches a lower MSE for the normally operating control valves than ANNs. However, the scope of the evaluated time series is too small to state, whether ridge regression could perform equally well when modelling non-linear behaviour of the control valves.

6.3. Use Case 2

The goal of the second use case was to experiment with data that has distortions due to an open manual bypass valve that is not stored in the data historian. It has not been analysed by industry experts, therefore, there is no information about why the manual bypass valves were opened. The predictions as well as the measured values are plotted in Figure 6.14 and the difference between the two lines in Figure 6.15. The error rate is extremely high when the manual bypass is open, however, the presence of early notifications is not as clear as it was in use case 1. There is a slow increase in the error rate that eventually results in large spikes several days before the opening of the manual bypass valves, however, the error rate decreases to negligible values prior to the actual opening. Therefore, it is unclear whether the opening of the manual bypass valves could have been reliably predicted beforehand. It was most probably caused by regular maintenance near the control valve, because the manual bypass valve is opened several times over the course of almost 2 months. This use case illustrates the complexity of petrochemical installations as experienced experts in the industry were not able to identify what forced the operators of the refinery to open the manual bypass valve without extensive investigation of the data.

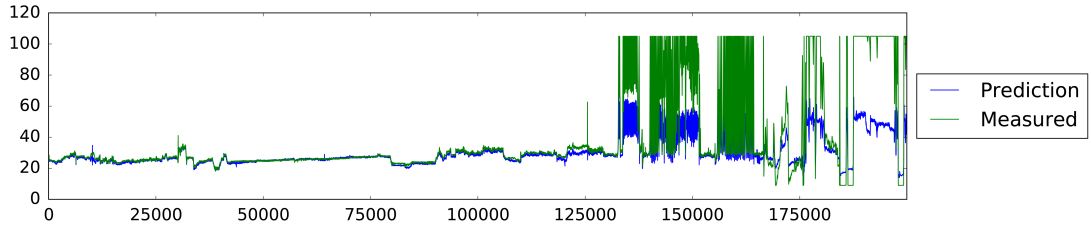


Figure 6.14: Actual and predicted values over a period of 4 months

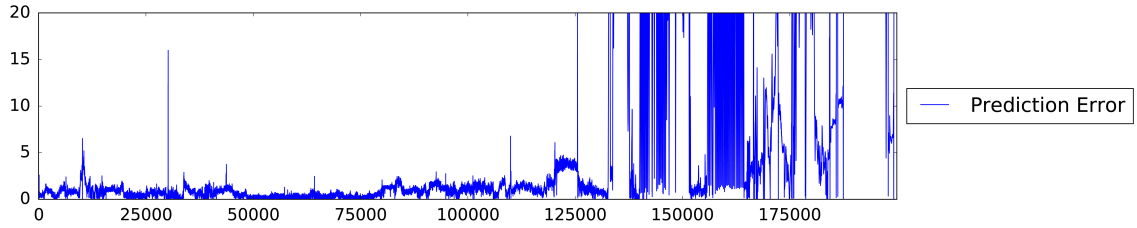


Figure 6.15: Difference between the actual and predicted values for over a period of 4 months

6.4. Normal Data Selection

In addition to the use cases, it was evaluated whether ridge regression could be used for normal data selection. Ridge regression does not overfit to non-linearities as much as neural networks, therefore, it should be possible to train the model using the same dataset that is used for predictions. Then, a certain portion of the data that might be anomalous can be discarded from the training set of the eventual prediction model. However, it is undefined what portion of the dataset is normal or abnormal. Ideally, the entire dataset should be normal, however, depending on the monitored system and timeframe it might also represent more than 10% of the entire dataset. Therefore, a compromise has to be found between the amount of normal data that is potentially discarded and the amount of abnormal data that might be kept in the dataset. In general, a certain percentile of the dataset with the lowest MSE could be kept or examples that have an error rate that is several times higher than the median could be discarded in order to automatically select the normal data points for the training set. The benefit of the fixed percentile approach is that the examples with the highest error rates are always discarded, however, it might result in a large portion of data that is discarded although it is perfectly correct. On the other hand, the discarding of the training examples based on the difference between the error rate and the median error rate enables better separation of values that are normal and definitely abnormal.

Normal data selection based on the ratio between the error rate and the median error rate is presented for two times series in Figure 6.16 and Figure 6.17. Similarly, the results have been plotted when using a fixed percentile in Figure 6.18 and Figure 6.19. The graphs present the actual values as well as the predicted values, based on the difference of which the abnormal training examples have been marked in blue. However, only the high-level distribution of the abnormal training examples is presented due to the low resolution of the images.

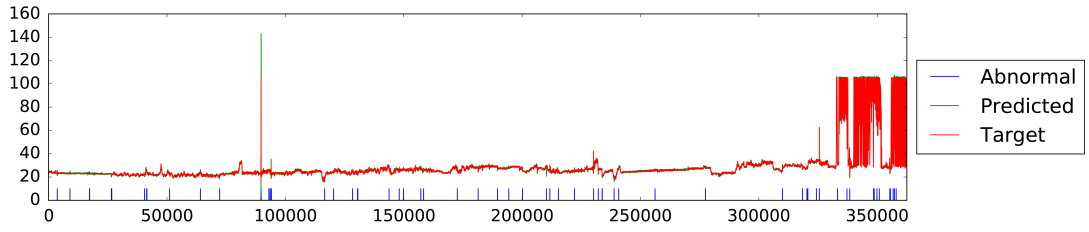


Figure 6.16: Normal data selection: all training examples with an error rate higher than 10x median are labelled as abnormal

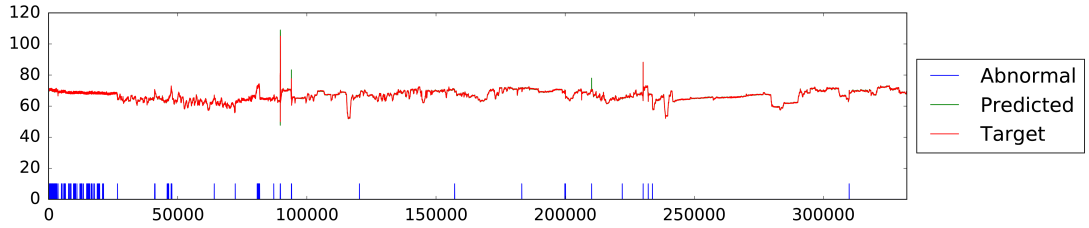


Figure 6.17: Normal data selection: all training examples with an error rate higher than 10x median are labelled as abnormal

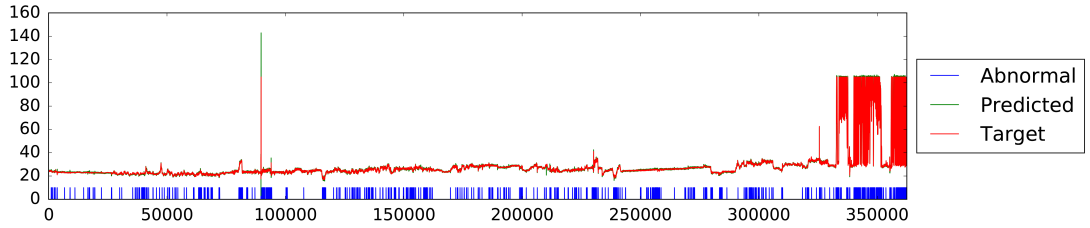


Figure 6.18: Normal data selection: 1% of the training examples with the highest error rate have been labelled as abnormal

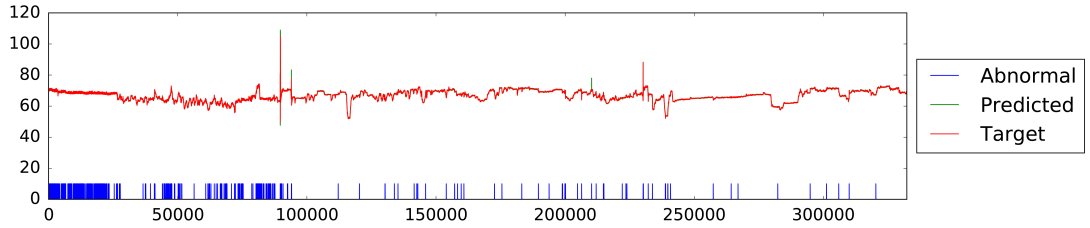


Figure 6.19: Normal data selection: 1% of the training examples with the highest error rate have been labelled as abnormal

The solutions presented in Figure 6.16 and Figure 6.17 are capable of determining the regions that actually have abnormal training examples. Furthermore, the concentration of the labels is very good as the training examples labelled as abnormal are more than a week apart from each other. This approach is capable of determining the regions with the highest error rates, however, it does not label all of the training examples that should be discarded as abnormal. Therefore, the sequence of training examples that will be discarded should be extended with values prior and after the label as well as based on the density of the labels.

The approach that is based on the percentile is capable of discarding almost all of the abnormal values caused by the opening of the manual valve. However, the distribution of the values labelled as abnormal is not as concentrated as it was when detecting the abnormal values using the previous method.

In general, both methods are capable of labelling regions of data in the training set that are potentially abnormal. Normal values will always be discarded when performing this type of data selection automatically as the threshold has to be set low enough to label most of the abnormal values. However, even the loss of a few percent of the normal data outweighs the distortions caused by abnormal data that is used in the training set.

6.5. Scalability

It is preferable to train the model on a single compute node whenever possible, as the parallelisation methods introduce additional complexity into the implementation. Therefore, performance of the different node types available via Databricks has been evaluated, as presented in Figure 6.20. The amount of vCPUs was selected to be the same for all CPU-based nodes while reducing the cost difference between the GPU and the CPU nodes. AWS I2 instances had the longest execution times when training 4-layer RNNs with 256 units and a historical length of 4. Therefore, it was selected as the baseline for comparison with other node types. AWS I2, R3 and C3 nodes are optimized for different tasks, however, their performance when training RNNs results in almost identical execution times. AWS C3 instances that are optimized for compute-expensive tasks reached a speed-up of less than 7% when compared to the I2 nodes. However, switching from CPUs to GPUs obtained a speed-up of almost 200%.

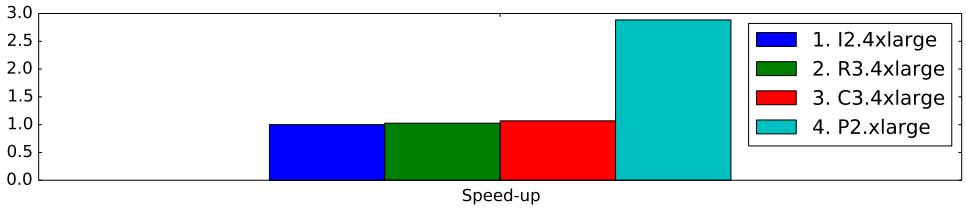


Figure 6.20: Comparison of the speed-up achieved from using different AWS EC2 instance types

The amount of data used for the training of machine learning models in this project was low enough for the training time to be less than 30 minutes for the best RNN configuration. Therefore, it was not required to utilize model parallelism nor data parallelism for speeding up the training phase. However, it is beneficial to distribute the tuning of the hyperparameters across a computer cluster, as a large number of hyperparameter combinations have to be tested in order to find the best solution.

7

Conclusions & Future Work

The objective of this research project was to evaluate whether machine learning methods exist that enable the creation of scalable models based on the normal operating conditions of a refinery that are capable of providing early fault predictions, and ultimately predictive maintenance solutions. The results of this project indicate that it is possible to reliably detect slow changes in the system that eventually cause issues in the refinery without any industry-specific knowledge injected into the machine learning approach. Furthermore, the method was able to detect the specific control valve that was behaving abnormally already more than a month before the accident. However, it is impossible to state whether a single model is capable of detecting all types of failures, because the amount of use cases within the battery limits of the modelled section of the refinery is low and identical failures rarely occur twice.

4-layer GRUs with *tanh* activation functions and an input sequence length of 4 samples provided the best results. Therefore, machine learning methods that model the behaviour of the refinery using time series data, e.g. RNNs, provide superior results to methods using discrete samples as inputs. However, it was presumed prior to the research that RNNs would be capable of performing far better than DNNs and ridge regression as the operation of refineries is inherently time-dependent. This is most probably caused by the low sampling rate as the measured values are stored once per minute while the closed-loop controllers operate once per second. Furthermore, the battery limits of the modelled subsystem should have been larger in order to model dependencies between the input features that are more than 10 minutes apart, as it is impossible to model dependencies that appear within a minute of each other due to the low sampling rate. Finally, it was considered to smooth the measured values, i.e. values denoted with *.PV*, but it was not performed as this would have filtered out information about failures caused by fast transitions, e.g. wiring issues. An example timeframe from a flow controller is presented in Figure 7.1, which clearly indicates high-frequency noise overlaying the lower-frequency signal. However, it might be reasonable to smooth the measured values in the future, if it enables the creation of a model that is capable of predicting the slower transitions in the system more reliably while neglecting the failures caused by fast transitions.

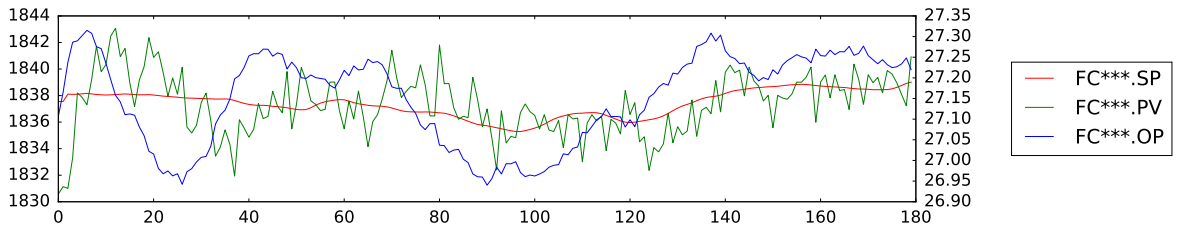


Figure 7.1: Operation of a flow controller during a 3h timeframe

Nevertheless, GRUs with *tanh* activation functions were able to outperform ridge regression and DNNs. Furthermore, GRUs required 7% less training time than LSTMs while reducing the MSE for the normal

operating conditions by more than 15%. The approach was not capable of reliably detecting an issue prior to the opening of the manual bypass valves. However, the MSE for use case 1 was less than 3% during normal operating conditions, while reaching more than 15% prior to the failure. Therefore, the model was capable of detecting the failure more than a month in advance due to an increase in the MSE by several times.

Therefore, the results of the research questions formulated in Chapter 1 can be summarized as follows:

1. It is possible to predict certain failures in petrochemical refineries using machine learning approaches based on the process control data for the normal operating conditions, however, it is impossible to evaluate the approaches for a wide variety of failures due to the limited availability of use cases.

2. The prediction horizon depends on the use case, however, the machine learning models developed for the main use case of this research project were able to predict the failure more than a month in advance.

3. RNNs that use time series data as inputs outperformed methods that use discrete values as inputs, however, the difference between the methods was smaller than expected prior to the research.

This research project provided a solid foundation for the development of early fault prediction and predictive maintenance solutions at Shell. However, the current research should be continued in several ways in order to fulfil the wider objectives of the organization:

1. *Prediction Horizon.* The most important aspect of early fault prediction and predictive maintenance that is currently missing is the prediction horizon. Early prediction of potential faults is a significant improvement for the operators of the refinery, however, the main question that arises for the operators is whether a component can last until the next turnaround even when it is behaving abnormally.

2. *Scope.* The current scope should be extended from 150 features to at least 1500 features. Additionally, the training set should be increased from a single year to all data available.

3. *Multiple Output Model.* The current implementation requires the creation of multiple models that have exactly the same inputs but a single output that differs. In practice, only a single model should be created that is capable of predicting a vector of values that can then be used to detect anomalies.

4. *Maintenance Data.* The current implementation is based on the process control data that is stored in the data historian. However, there is a significant portion of data available about the maintenance of the refinery. The possibility of integrating that dataset with the current implementation should be evaluated, as this could provide the operators with information about prior maintenance for similar use cases.

5. *Artificial Data.* Evaluation of the current implementation is largely limited by the number of available use cases in the modelled section of the refinery. Artificial datasets could be crafted based on industry-specific knowledge of potential failures in order to increase the number of potential failures against which the implementation can be verified.

A

Naming Conventions

Petrochemical refineries are composed of tens of thousands of components which have to be uniquely identified, therefore, a unique name is assigned to all of the components. The parameter names are split into multiple sections, e.g. *XYZ:***FC***.SP* as indicated in Chapter 2, which can be described as follows:

1. *XYZ*. The first letters of the name indicate the high-level plant to which the component has been assigned. A single plant might be constructed of thousands of different components which frequently yield more than 5,000 measured parameters. All of the measurements in this research project were obtained from a single plant that was anonymized throughout the text using *XYZ* due to confidentiality.

2. *****. The numbers directly after the colon indicate the lower-level plant to which a component has been assigned. All of the measurements in this research project were obtained from a single lower-level plant that was anonymized throughout the text using ***** due to confidentiality.

3. *FC*. The unit description is provided after the plant name and code. This name refers to a specific type of device used throughout the refinery. It includes a range of application areas, where flow controllers (FC), flow indicators (FI) and flow transmitters (FT) are units commonly related to flow characteristics.

4. *****. Many units with the same type might be assigned to a single plant, therefore, each unit is uniquely identified using a code.

5. *SP*. Finally, the stored parameters of a unit are described, e.g. the process value (PV) that represents the measurements, setpoint (SP) that defines the desired values and the output (OP) that is used to control the PV in order to obtain the desired SP.

The units and their types vary greatly among different plants and sections of a plant, therefore, this appendix provides an overview of only the most common units that were used in the section of the refinery that was modelled during this research project.

Bibliography

[1] Deep visualization toolbox. https://github.com/yosinski/deep-visualization-toolbox, Accessed: June 20, 2017.

[2] Tensorflow api documentation. https://www.tensorflow.org/versions/r0.11/api_docs/python/nn/activation_functions_, Accessed: June 20, 2017.

[3] Cs231n convolutional neural networks for visual recogniion. http://cs231n.github.io/neural-networks-1/, Accessed: June 20, 2017.

[4] Autoencoder. https://en.wikipedia.org/wiki/Autoencoder, Accessed: June 20, 2017.

[5] Cs231n convolutional neural networks for visual recogniion. http://cs231n.github.io/convolutional-networks/, Accessed: June 20, 2017.

[6] Cs231n convolutional neural networks for visual recogniion. http://cs231n.github.io/understanding-cnn/, Accessed: June 20, 2017.

[7] Cs231n convolutional neural networks for visual recogniion. http://cs231n.github.io/neural-networks-2/, Accessed: June 20, 2017.

[8] Connected barrels: Transforming oil and gas strategies with the internet of things. https://dupress.deloitte.com/dup-us-en/focus/internet-of-things/iot-in-oil-and-gas-industry.html, Accessed: June 20, 2017.

[9] Control valve sourcebook — refining. http://www.documentation.emersonprocess.com/groups/public/documents/book/d103205x012.pdf, Accessed: June 20, 2017.

[10] Nvidia tesla k80. http://www.nvidia.com/object/tesla-k80.html, Accessed: June 20, 2017.

[11] Lasso (statistics). https://en.wikipedia.org/wiki/Lasso_(statistics), Accessed: June 20, 2017.

[12] Linear regression. https://en.wikipedia.org/wiki/Linear_regression, Accessed: June 20, 2017.

[13] Long short-term memory. https://en.wikipedia.org/wiki/Long_short-term_memory, Accessed: June 20, 2017.

[14] Polynomial regression. https://en.wikipedia.org/wiki/Polynomial_regression, Accessed: June 20, 2017.

[15] Regularization: Ridge regression and the lasso. http://statweb.stanford.edu/ tibs/sta305files/Rudyregularization.pdf, Accessed: June 20, 2017.

[16] Tikhonov regularization. https://en.wikipedia.org/wiki/Tikhonov_regularization, Accessed: June 20, 2017.

[17] Shell pernis. http://s03.static-shell.com/content/dam/royaldutchshell/documents/corporate/shell-pernis-infographic.pdf, Accessed: June 20, 2017.

[18] Apache spark. https://spark.apache.org/, Accessed: June 20, 2017.

[19] Control valves and flow characteristics. http://www.engineeringtoolbox.com/control-valves-flow-characteristics-d_485.html, Accessed: June 20, 2017.

[20] T. Wang D. J. Wu A: Coates, B. Huval and A. Y. Ng. Deep learning with cots hpc systems. 2013.

[21] J. Shawe-Taylor A. J. Smola B. Schölkopf, J. C. Platt and R. C. Williamson. Estimating the support of a high-dimensional distribution. 1999.

[22] P. Bangalore and L. B. Tjernberg. An artificial neural network approach for early fault detection of gearbox bearings. *IEEE Transactions on Smart Grid*, 6, 2015. ISSN 1949-3061.

[23] D.W.B. Bender. Early fault detection in industrial plants. Master's thesis, Delft University of Technology, 2016.

[24] Y. Lin-Y. Yu G. Bradski C. Chu, S. K. Kim and A. Y. Ng. Map-reduce for machine learning on multicore. *Advances in Neural Information Processing Systems*, 19, 2006.

[25] D. S. Yeung D. Wang and E. C. C. Tsang. Structured one-class classification. *IEEE Transactions on Systems, Man, and Cybernetics*, 36, 2006. ISSN 1941-0492.

[26] J. Dean and U. Hölzle. Build and train machine learning models on our new google cloud tpus. https://blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/, Accessed: June 20, 2017.

[27] S. O. Haykin. *Neural Networks and Learning Machines*. Pearson, 2009.

[28] Y. Bengio I. Goodfellow and A. Courville. *Deep Learning*. MIT Press, 2016.

[29] K. Cho J. Chung, C. Gulcehre and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. 2014.

[30] et al. Jouppi. In-datacenter performance analysis of a tensor processing unit. 2017.

[31] N. Jouppi. Google supercharges machine learning tasks with tpu custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html, Accessed: June 20, 2017.

[32] A. D. Keromytis K. A. Heller, K. M. Svore and S. J. Stolfo. One class support vector machines for detecting anomalous windows registry accesses. 2003.

[33] R. K. Srivastava K. Greff and J. Koutník. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, PP, 2016. ISSN 2162-2388.

[34] Q. Xiong M. Fan G. Sun L. Ma K. Wang, Y. Zhao and T. Liu. Research on healthy anomaly detection model based on deep learning from multiple time-series physiological signals. *Scientific Programming*, 2016, 2016.

[35] T. Cohn K. Yao, K. Vylomova and C. Dyer. Depth-gated recurrent neural networks. 2015.

[36] J. Keuper and F. Pfreundt. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. 2016.

[37] J. McDermott L. Bontemps, V. L. Cao and N. Le-Khac. Collective anomaly detection based on long short term memory recurrent neural network. 2017.

[38] J. M. Molina L. Martí, N. Sanchez-Pi and A. C. B. Garcia. Anomaly detection based on sensor data in petroleum industry applications. 2015. ISSN 1424-8220.

[39] O. Lyudchik. Outlier detection using autoencoders. 2016.

[40] A. Nanduri and L. Sherry. Anomaly detection in aircraft data using recurrent neural networks (rnn). *Integrated Communications Navigation and Surveillance*, 2016.

[41] J. Novet. Intel will test nervana's 'lake crest' silicon in first half of 2017, 'knights crest' also coming. https://venturebeat.com/2016/11/17/intel-will-test-nervanas-lake-crest-silicon-in-first-half-of-2017-knights-crest-also-coming/, Accessed: June 20, 2017.

[42] C. Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, Accessed: June 20, 2017.

[43] J. Liu S. Yu P. Jiang, Z. Hu and F. Wu. Fault diagnosis based on chemical sensor data with an active deep neural network. 2016.

[44] J. Panettieri. Cloud market share 2017. https://www.channele2e.com/2017/02/09/cloud-market-share-2017-amazon-microsoft-ibm-google/, Accessed: June 20, 2017.

[45] W. Zaremba R. Jozefowicz and I. Sutskever. An empirical exploration of recurrent network architectures. 2015.

[46] Y. Lan R. Zhang, S. Zhang and J. Jiang. Network anomaly detection using one class support vector machine. 2008.

[47] J. Wang R. Zhao, R. Yan and K. Mao. Learning to monitor machine health with convolutional bi-directional lstm networks. 2017.

[48] C. Scheibböck S. Dreiseitl, M. Osl and M. Binder. Outlier detection with one-class svms: An application to melanoma prognosis. 2010.

[49] S. Karunasekera S. M. Erfani, S. Rajasegarar and C. Leckie. High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. 2016.

[50] K. Sato. Hadoop conference japan 2016. http://www.systranet.com/turl/?systrangui=rss&systranuid=aHR0cC1xaWl0YS5jb20va2F6dW5vcmkyNzkvaXRlbXMvOTgxYThhMmE0NGY1ZDExNzI4NTYvamFfZW4

[51] A. Singh. Applying artificial intelligence in medicine: Our early results. https://blog.cardiogr.am/applying-artificial-intelligence-in-medicine-our-early-results-78bfe7605d32, Accessed: June 20, 2017.

[52] T. C. Clancy T. J. O'Shea and R. W. McGwier. Recurrent neural radio anomaly detection. 2016.

[53] D. M. J. Tax and R. P. W. Duin. Support vector data description. 2004. ISSN 1573-0565.

[54] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. ELSEVIER SCIENCE & TECHNOLOGY, 2008.

[55] C. Garrigues V. Garcia-Font and H. Rifà-Pous. A comparative study of anomaly detection techniques for smart city wireless sensor networks. 2016.

[56] J. Vincent. Arm's new chip design focuses on ai and machine learning. https://www.theverge.com/2017/3/21/14998100/arm-new-dynamiq-microarchitecture-ai-chip-design, Accessed: June 20, 2017.