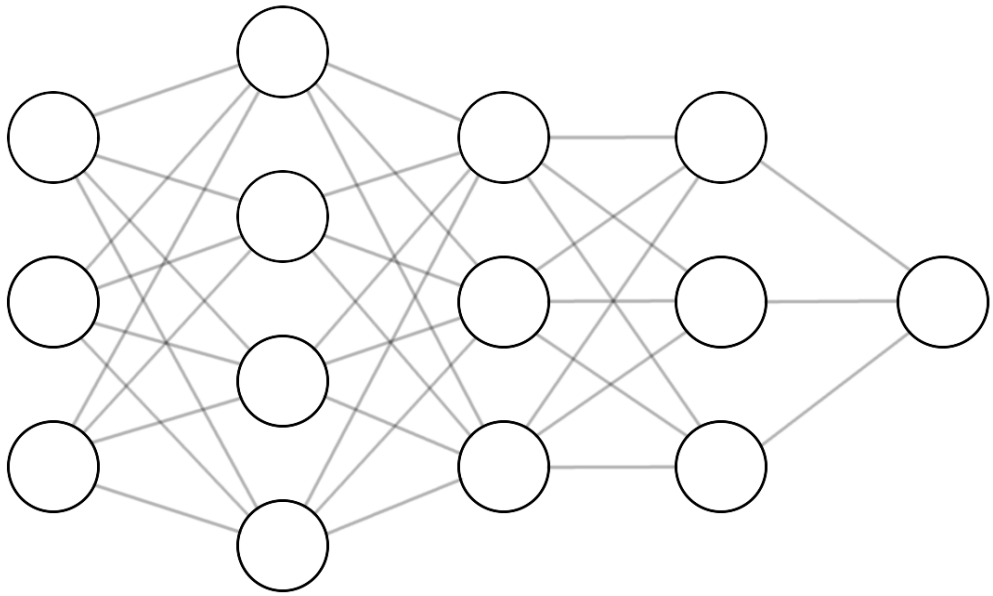

Software life cycle management
with supervised learning

Oscar Andersson
Erik Thorsson Högfeldt

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Software life cycle management with supervised learning

Oscar Andersson
Erik Thorsson Högfeldt

Spring 2017

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Software life cycle management with supervised learning
Oscar Andersson
Erik Thorsson Högfeldt

Examiner: Peter Lundin
Department of Computer Science and Engineering
Chalmers University of Technology / University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
A visual representation of an artificial neural network.

Department of Computer Science and Engineering
Gotheburg 2017

Abstract

The field of machine learning have had an upswing in popularity in the recent years. Computation of complex neural networks, previously not applicable due to hardware restraints, have been made more viable with recent advancements in GPU-acceleration technology.

Software life cycle management is the administration of the cyclic software development process involving planning, building, testing and publishing.

The purpose of this thesis was to investigate if supervised learning, a type of machine learning task, can be used as an useful tool for software life cycle management. The goals were to develop machine learning software capable of analysing vehicle data, which could bring additional information about faults. The thesis presents the machine learning methods and strategies used to construct and optimise the software.

The software created can recognise faults in data resembling data collected from cars' electrical system and classify which faults. The potential of analysing vehicle data with supervised learning models is acknowledged during a discussion section along with a proposition for further application with real world vehicle data.

Keywords: Machine learning, supervised learning, neural networks, data.

Software life cycle management with supervised learning

Oscar Andersson
Erik Thorsson Högfeldt
Examinator: Peter Lundin
Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet

Examensarbete

Sammanfattning

Maskininlärning är ett område som har ökat i popularitet under de senaste åren. Beräkning av komplexa neurala nätverk, som tidigare var opraktiska på grund av hårdvarurestriktioner, har blivit enklare att genomföra tack vare de senaste framstegen inom GPU-accelerationsteknologi.

Software life cycle management är hanteringen av den cykliska mjukvaruutvecklingsprocessen som innefattar planering, byggande, testning och publicering.

Syftet med detta arbete var att undersöka om supervised learning, en typ av maskininlärningsuppgift, kan användas som ett användbart verktyg för software life cycle management. Målsättningen var att utveckla maskininlärnings-mjukvara som kan analysera fordonsdata, vilket kan ge ytterligare information om fel. Rapporten presenterar maskininlärningsmetoder och strategier som används för att konstruera och optimera denna mjukvaran.

Den skapade mjukvaran kan känna igen fel i data som ska efterlikna datan från bilars elsystem och kan specificera vilket fel det är. Möjligheten att analysera fordonsdata med supervised learning modeller utreds i diskussions avsnittet. Detta diskuteras tillsammans med förslag för vidare tillämpning med verklig fordonsdata.

Nyckelord: Maskininlärning, supervised learning, neurala nätverk, data.

Preface

This bachelor thesis is created for the Bachelor of Computer Science and Engineering program (180 credits), a part of the Department of Computer Science and Engineering at Chalmers University of Technology. The project covers 15 credits.

The work was done at Semcon Sweden AB.

We would like to thank our mentors at Semcon, Christer Ek and Peter Nordin, for the continued help during this project. We are also thankful to Christer Carlsson, our mentor at Chalmers University of Technology. Lastly, we would like to thank Benjamin Vedder for taking his time to gather the data used in this thesis.

Dictionary

machine learning model	A set of software components capable of learning from and making assumptions on data.
convergence rate	How fast a machine learning model reaches its optimal point.
hyperparameters	Parameters for configuring machine learning models.
learning rate	A hyperparameter for how large corrections optimisers perform.
data sample	A set of values defining an instance.
feature	A value corresponding to a specific attribute.
labels	Desired outputs from a machine learning model when facing a corresponding set of inputs.
cost	A quantitative measurement for a model's performance.
preprocessor	An operation that modifies data before usage.
epoch	Complete training series on all available data.
CLI	Short for Command Line Interface. An interface of which communication and interaction between users and computer programs are made through commands in the form of consecutive lines of text.

Contents

Introduction

1.1 Background

The electrical system in today's and upcoming cars can be monitored to provide detailed information about almost all aspects of the car. This large amount of data could potentially be used to detect and identify problems within the car. Problems that could possibly be found long before they are visible to the user.

The ability to find problems and make predictions based on data from the cars' electrical system could provide software and hardware developers with useful information to improve upcoming software and hardware. This is becoming increasingly more interesting as the software development often is iterative and as the deployment of new software becomes easier. Machine learning could potentially be used to do this by being able to learn without being explicitly programmed to.

Semcon Sweden AB wants to investigate the use of different machine learning tasks to extract useful information from cars' electrical systems.

This project is done as a part of the NGEA (Next Generation Electrical Architecture) project. The aim of the NGEA project is to improve the automotive industry by strengthening research involving connected cars to encourage a quicker and more flexible development.

1.2 Purpose

The purpose of this work is to explore the possibility of using supervised learning to improve the software life cycle in the automotive industry.

1.3 Goal

The main goal of this project is to create and train supervised learning models so they can be used to provide useful information from data logs containing either errors or a failure. The focus of the work will be programming working models to showcase if supervised learning would be applicable in the software life cycle. To achieve this, the following needs to be done:

- Program a parser to rearrange the provided data to work with the machine learning libraries and tools.
- Research supervised learning and develop models that best suits the data and the desired result.
- Present and evaluate the results from the models.

1.4 Delimitations

The data needed to train and test the supervised learning models will be provided by Semcon. This project will take advantage of pre-made tools and libraries to create, train and evaluate the supervised learning models. The main library to be used will be TensorFlow.

Introduction to machine learning

2.1 Machine learning

Machine learning is a study of computer science that enables computer programs to learn without being explicitly programmed. Programs utilising machine learning can improve their ability to perform tasks based on past experience. Machine learning algorithms iteratively learns from and can find patterns in the inserted data.

2.2 Types of machine learning systems

Machine learning systems can be classified in different categories based on the type and amount of supervision they are given, the accessibility of data they are given and how the systems proceed with learning.

2.2.1 Supervision

Machine learning systems can be classified by the type and amount of supervision they receive during training. Machine learning systems classified under supervised learning are trained with data containing desired outputs (also known as labels). Supervised systems train with examples containing a set of inputs and the desired outcome expected from the system. The system algorithms will attempt to minimise the error between the system's output and the desired output in order to improve the system's prediction accuracy. [1, p.8]

Machine learning systems classified under unsupervised learning are trained with data lacking desired outputs. Unsupervised systems train using only inputs and its algorithms learn by finding underlying structures of the training data. [2, p.52]

2.2.2 Model-based and instance-based learning

Model-based machine learning systems generalise by using training data to construct a model used to make predictions. The contrary for model-based learning is instance-based learning, which memorise examples from training data and generalise though comparing new instances with memorised instances. [1, p.17-18]

2.2.3 Batch-based and online-based learning

Machine learning systems using batch-based learning learn through training on all available data during a training phase and cannot learn after being deployed into production. Systems using online-based learning incrementally learns from smaller or even individual instances. A drawback of the batch-based learning systems is that in order to adapt to new training data, it has to retrain the entire system from the beginning with the new data and the old. Online-based systems can learn from new data as it arrives, increasing the knowledge already learned. [1, p.14-15]

2.3 Features

An attribute can be described as a measurable characteristic of an object or phenomenon such as the temperature of a certain sensor or the current running through a motor. In machine learning a feature is usually described as an attribute and its value, for example $sensorTemperature = 21$ or $motorCurrent = 2$.

2.3.1 The curse of dimensionality

Learning from data become increasingly more difficult as the amount of features increase. The higher dimensionality of data processed, the more instances of data are needed to recognise a pattern in the dimensional space. [3, p.22-23, 649]

Quote from "The Elements of Statistical Learning" illustrating a manifestation of the curse of dimensionality:

> "Another manifestation of the curse is that the sampling density is proportional to N1/p, where p is the dimension of the input space and N is the sample size. Thus, if N1 = 100 represents a dense sample for a single input problem, then N10 = 10010 is the sample size required for the same sampling density with 10 inputs. Thus in high dimensions all feasible training samples sparsely populate the input space."[3, p.23]

2.3.2 Feature engineering

The selected sets of features used as input can have a great effect on the machine learning systems' ability to learn. The process of selecting the right features, combining features and adding more features from new data is called feature engineering. The feature engineering process is an important step during machine learning projects. [1, p.25-26]

Feature selection

Feature selection is the process of selecting the most fruitful features for constructing a useful model. The feature selection process also includes disregarding irrelevant features that would increase dimensionality without any significant benefit. [1, p.26]

Feature extraction

During feature extraction, new features are created by combining original features that become more informative and less redundant when derived into one. [1, p.26]

For an example of feature extraction, Machine learning models can be used in the stock market to estimate what shares are most likely to return profit. A share's percentage gain or loss over a day can be derived from the share's opening price and closing price by the equation: $percentage = (closingprice - openingprice)/(openingprice)$.

This would reduce the amount of dimensions while generating a feature that firmly indicates the success of a share.

Feature scaling

The range of values may vary significantly between different sets of features. For example, values logged from cars can have ranges of $0 < sensorTemperature < 100$ and $0 < throttleRatio < 10$.

Features with smaller value ranges might be neglected by machine learning algorithms in favour of features with broad ranges of values. Feature scaling is a method that either normalises or standardises the feature sets during data preprocessing, adjusting all feature values to scale within the same span, lessening the effects of divergent value ranges among feature sets.[1, p.65]

To better illustrate, here is an example two sets of features undergoing preprocessing with min-max normalisation:

Original feature sets	**Sets normalised with min-max scaling**
$sensorTemperature = [0, 40, 100]$	$sensorTemperature = [0.0, 0.4, 1.0]$
$throttleRatio = [0.0, 5.0, 10.0]$	$throttleRatio = [0.0, 0.5, 1.0]$

2.4 Supervised learning tasks

2.4.1 Regression

A common type of tasks in supervised learning are regression tasks. A regression task is to predict numeric values based on a set of given features as input. During training, regression algorithms models relationships between the features and the output. [1, p.8][3, p.10]

2.4.2 Classification

Classification tasks are similar to regression tasks with the exception that the model outputs predictions on a set of categories, e.g predicting what type of fault a car has by analysing its log data.[1, p.8][3, p.10]

2.5 Artificial Neural Networks

Artificial neural networks are models that have extensive amounts of artificial neurons connected with each other through layers, inspired by how the neural networks of biological brains are structured. Artificial neurons are nodes in an artificial neural network that upon receiving a number of inputs, outputs values based on the neurons' activation functions. The first layer

of an artificial neural network is called the input layer, which receives the data inserted to the network. The output layer is the last layer of an artificial neural network, which neurons emit the final output of the network. Artificial neural networks can also have hidden layers, placed amidst the input and output layers, which increase the complexity of what the network is capable of output. See Figure 2.1 for a representation of an artificial neural network. [2, p.33-36]

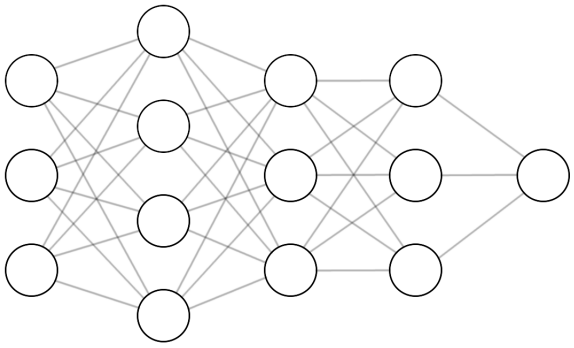

Figure 2.1: Illustration of a neural network

2.5.1 Propagation functions

The propagation function of a neuron transforms the outputs received from other neurons to scalar input which is used by the activation function. [2, p.35]

2.5.2 Activation functions

Each neuron of an artificial neural network has an activation function that outputs a reaction based on the input from the propagation function. There are various types of activation functions that calculate the input differently resulting in different output characteristics, see Figure 2.2 for plotted activation functions. [2, p.36]

Common activation functions used in artificial neural networks:

- Logistic (Referred as *Sigmoid* in the context of TensorFlow)

$$f(x) = \frac{1}{(1 + e^{-x})}$$

- TanH

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

- ReLU

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

- ELU

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

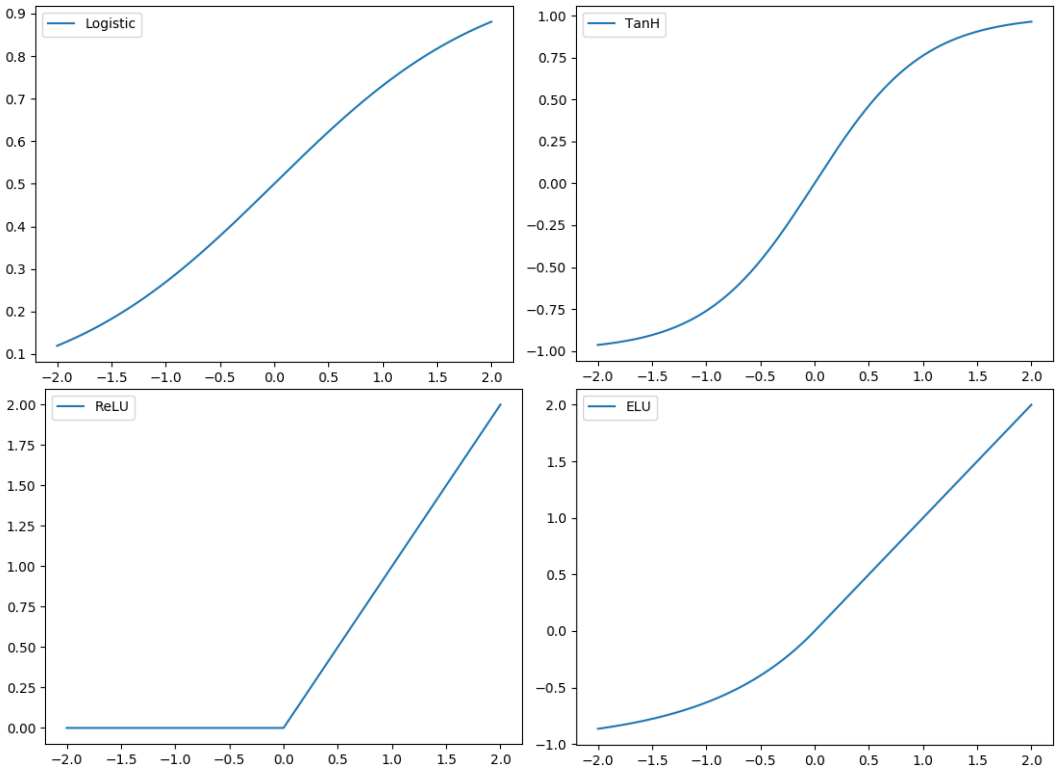

Figure 2.2: Plotted activation functions

2.5.3 Output functions

The output function of a neuron calculates the values that are distributed to the neurons connected to its output, based on the neuron's activation function. Often, the output from the activation function is directly used as output by the output function. [2, p.38]

2.5.4 Feedforward Neural Networks

Artificial neural networks can have different interconnection patterns between its neurons. In feedforward neural networks, information flow in only one direction. Neurons of any layer cannot output information to any other layers than the next layer towards the output layer. [2, p.39]

2.6 Weights and biases

The signals between neurons in an artificial neural network can be modified using weights which magnifies or compress connections between neurons. During training, the weights of artificial neural networks are individually modified in order to achieve desired output from the networks' output layer. The weights are applied by the propagation function. Applying bias to an activation function shifts its curve along the x-axis changing the characteristics of the output, which can be used for tuning network output. A visualisation of the effects of biases and weights can be seen in Figure 2.3. [2, p.34,44]

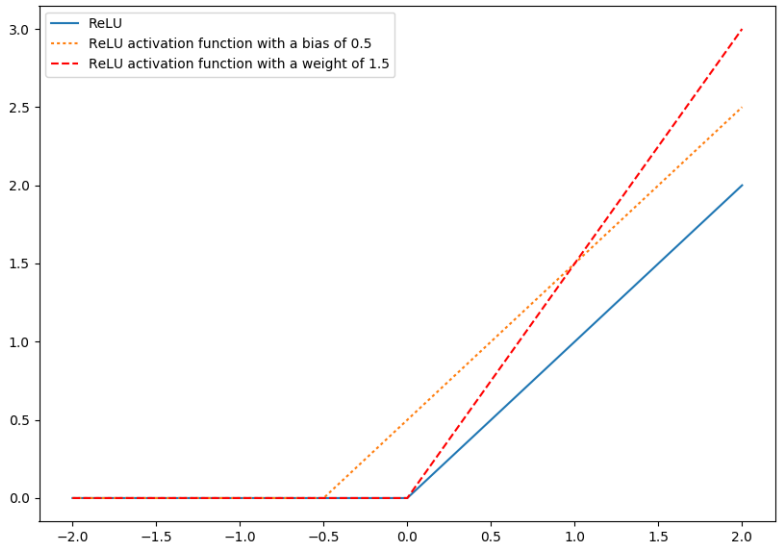

Figure 2.3: Graph demonstrating how bias and weight effects an output of a neuron

2.7 Performance measuring functions

Performance measuring functions are used to calculate machine learning models efficiency during evaluations and optimisation. Utility and cost functions measure the performance of the model. [1, p.37]

Common cost functions used:

- Root Mean Square Error

$$RMSE(X,h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2}$$

- Mean Absolute Error

$$MAE(X,h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(x^{(i)}\right) - y^{(i)}\right|$$

- Mean reduced Softmax cross entropy

$$softmax.C.E(X,h) = -\frac{1}{m}\sum_{i=1}^{m}\left((x^{(i)}log(h(x^{(i)})) + (1-x^{(i)})log(1-h(x^{(i)}))\right)$$

- Mean reduced Sigmoid cross entropy

$$sigmoid.C.E(X,h) = \frac{1}{m}\sum_{i=1}^{m}\left(max(x^{(i)},0) - x^{(i)}h(x^{(i)}) + log(1+exp(-abs(x^{(i)})))\right)$$

X = Matrix of all feature values
x^i = A vector containing all feature values of instance i
y^i = A vector containing the label of instance i
h = The system's hypothesis, the system's prediction
m = The number of instances in data set being evaluated

2.8 Optimisation algorithms

Optimisation algorithms, in the context of machine learning, modify the weights and biases of models in order to minimise the cost function. Most optimisation algorithms have configurable parameters that regulate how momentous its corrections are.

2.8.1 Gradient descent

Gradient descent is an iterative optimisation algorithm performing gradual network adjustments to minimise the cost emitted from performance measuring functions. The algorithm will gradually attempt to decrease costs until it reaches a point of convergence, the point of which it cannot further decrease the cost.

One significant drawback of the gradient descent algorithm is that it is prone to get stuck on local minimums. It presumes that it have reached the lowest cost it can achieve, but there are weight and bias arrangements of the network that can render lower costs. The cause of these pitfalls is that adjustments made at each step are too minuscule [1, p.112]. The steps are not large enough to scale the peaks between local and global minimums, as illustrated by Figure 2.4.

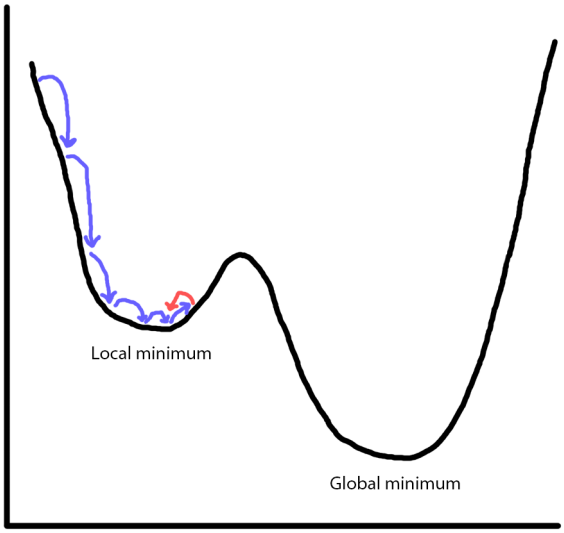

Figure 2.4: An example of a gradient descent pitfall
x: parameter vector | y: cost

2.8.2 Momentum

Momentum is based on gradient descent but additionally utilises an exponentially decaying average of the previous gradients to gain a momentum effect. The momentum effect lessens chances of getting stuck at local minimums and improves the rate of which the models converge. [1, p.294]

2.8.3 Adam

Short for adaptive moment estimation, is an optimisation algorithm that achieves adaptive learning rates in a similar manner to the Momentum algorithm. It uses an exponentially decaying average of past gradients like Momentum, but in conjunction uses an exponentially decaying average of past squared gradients. Adam is considered to be one of the best adaptive learning algorithms for machine learning at the time of writing this report [1, p.293,299][4, p.6-10]. [4]

Method and tools

In this chapter, the methodology used in the project will be explained.

3.1 Tools

This section lists and describes various tools used in the project.

3.1.1 TensorFlow

TensorFlow is an machine learning library that enables construction of machine learning models expressed through code written in conventional programming languages.

Python language

The TensorFlow API can be used in several programming languages such as Java, C++, Go and Python. The API support for python is currently much stronger than the others due to it being the most complete, easiest to use and most stable version according to the TensorFlow creators themselves [5]. The Python API also has more comprehensive documentation than its counterparts.

CUDA acceleration

One significant feature of TensorFlow is the support for GPU-acceleration with NVIDIA CUDA Deep Neural Network library[6]. A large collection of operations can be greatly GPU-accelerated due to them having exceptional parallel computing capabilities. By offloading tasks which scales well with parallel computing, the training time of machine learning models can be greatly reduced. See Figure A.1 for benchmarks.

TensorBoard

TensorFlow has support for logging variables and operations, called events, during sessions to event files. The content of these files can be visualised by the tool TensorBoard. TensorBoard can plot events and draw histograms for complex data. Graphs of the models' structure and components can also be rendered. TensorBoard is a useful tool for understanding models and for troubleshooting seemingly incomprehensible behaviour. TensorBoard services hosts the websites which acts as the interface for navigating different visualisations.

3.2 Project phases

The progress of the project can be divided into four main phases:

I. Researching the computer science of machine learning and how to use TensorFlow to construct machine learning models.

II. Create software to parse and preprocess data.

   i) The data variables are to be extracted from comma separated variable files and parsed accordingly.

   ii) Create function for selecting the desired features out of the parsed data.

   iii) Create function for extracting labels out of the parsed data.

   iv) In order to obtain a memory functionality for the machine learning models, the ability to introduce additional time deltas, i.e. including features from earlier data samples, should be created.

III. Construct machine learning models.

   i) Constructing neural networks.

   ii) Selecting cost function and optimisation functions.

   iii) Implement batch normalisation.

IV. Refinement stage.

   i) Examine model performance with different amounts of hidden layers and neurons.

   ii) Evaluate layer components performance and investigate alternatives.

   iii) Implementing regularisation techniques.

   iv) Implement TensorBoard to improve troubleshooting.

3.3 Method

The initial part of the project will be to gather and parse data that would later be used in the machine learning models. The data to be used by the models will not be available at the start of the project. To work around this simulated data will be used from a pre-made car transmission model in Simulink.

The parser will be developed in Python and needs to be able to read desired values from CSV-files. The different values also needs to be grouped into features and labels. This will be achieved using standard Python libraries and NumPy, a Python library that includes array and matrix related functions. A basic regression-based supervised learning model will be created using the simulated data. The model will be created using TensorFlow and programmed in Python.

During the development of the supervised learning model, testing and evaluation will be done on private computer hardware with support for CUDA acceleration. TensorBoard will serve as an aid for debugging and evaluating the model.

To explore the different supervised learning tasks two models will be created. The fundamental parts of both models will be shared but one will be regression-based and the other will be classification-based. The regression model will require one of the data values to be used as the label. This label would preferably be dependent upon all other features. The classification model will require the original data to be manually labelled as either faulty or working. The parser needs to be modified to use the manually labelled data. During optimisation of the models different machine learning model elements, such as optimisation functions and cost functions, will be evaluated.

When the provided data becomes available the parser and the models would be changed to accommodate the new data. These changes will heavily depend upon the structure and complexity of the data. The final models will be evaluated with parts of the provided data reserved for testing. The results of the final models will be compiled into graphs to showcase the abilities of the models.

Implementation

This chapter describes the decision and solutions made during the project.

4.1 Simulation data

In order to compensate for the lack of genuine vehicle data during the start of the project, the initial data used to test constructed parsers and machine learning models was simulated. Simulations were achieved using the Simulink model *Parallel Hybrid Transmission* [7].

Functions that simulates the environmental influence of the vehicle, such as wind resistance and road incline, was kept with the default configuration due to satisfactory standard setup. Parameters were set to regulate how and when the vehicle accelerates and stops. This was done to establish a simulation sequence that was longer and more varying.

Signals of the simulation's electrical system was logged and exported to CSV-files where each row represents one time sample.

4.2 Data parsers and preprocessors

The content of the CSV-files' need to be transformed to an array data structure in order to be used by models constructed with TensorFlow. This was achieved by creating a function that iterate through each row of the file, splits the comma separated variables and appends them to an array. (See Appendix B, Figure B.1, for an illustration of the function's operational principle.)

The ability to select which features to use from the data array is mandatory in order to perform feature selection. Therefore, the NumPy function *delete* was utilised for removing unwanted features from arrays. Figure B.2 shows a typical implementation of *NumPy.delete*.

Adding features from samples of earlier time deltas to preceding samples gives the machine learning model a sense of memory of what have happened in the past. This was done with a function that iterates through all samples of an array and stacks the features of earlier samples to the current sample. The first samples of an array that have insufficient amounts of previous samples have to be discarded. See Figure B.3.

The machine learning models in this project benefits from training on small batches of samples instead of all training data at each training step [8, p.2]. In order to serve batches to machine learning models, a function that strategically selects parsed data and sends it as a batch was implemented. See Figure B.4 for a classification batch fetcher and Figure B.5 for a regression batch fetcher.

4.3 Initial construction of the machine learning models

The initial construction of the machine learning models will be discussed in this section. The choice of initial network structure, performance measurements, optimisers and session environments will be presented and motivated.

Code samples presented in this section will have the TensorFlow library imported as *tf*.

4.3.1 Placeholders

In order to pass data through the neural network, so called placeholders are needed. A placeholder, in the context of TensorFlow, is a variable that will be assigned data at a later stage. It enables creation of operations and construction of models without requiring the data to be available. During training and evaluation sessions, placeholders need to be fed with data before running any operations.

Example of placeholders for samples and labels:

```
X = tf.placeholder(dtype=tf.float32, name="SampleValues")
Y = tf.placeholder(dtype=tf.float32, name="LabelValues")
```

4.3.2 Forming artificial neural networks

Modelling the network using TensorFlow started with declaring the structure of which the layers and neurons would take.

Layer substance

The weights and biases of layers consists of variable arrangement in the form of float matrices. These weight and bias arrays are used by the layer's propagation functions to adjust neuron input. The arrays are initially filled with normal distributed random values, to speed up convergence rates.

```
layer_variables = {
'weights':tf.Variable(tf.truncated_normal(
    ([neuron_amount_previous_layer, neuron_amount_this_layer]) ),
'biases': tf.Variable( tf.truncated_normal([neuron_amount_this_layer]) )
}
```

The weights of the input layer exclusively process the input data, acting as an entry for the features to be inserted. The hidden and output layers receives input from their preceding layer.

Propagation functions applying weights and biases were implemented using TensorFlow matrix and arithmetic operators:

```
weighted_input = tf.matmul(output_previous_layer,
                           layer_x['weights'])

layer_propagation = tf.add(weighted_input, layer_x['biases'])
```

The initial activation function for all hidden layers in the neural network was ReLU, which is considered to be one of the most computation efficient activation function while generating preferable output [1, p.272]. It was chosen in assumption it was a good activation function to

establish the initial network with. Later on in the project, different activation functions were evaluated in comparison to ReLU (see subsection 4.6.2).

Implementation of a ReLU activation function with TensorFlow:

```
layer_activation = tf.nn.relu(layer_propagation)
```

The output function of each layer at the initial stage of modelling was just the activation functions output. An example containing the same layer setup as the models used:

```
# Post layer variables initialisation
# Layer 5
weighted_input_l5 = tf.matmul(layer_4_output,
                              layer_5['weights'])
layer_5_propagation = tf.add(weighted_input_l5, layer_5['biases'])
layer_5_activation = tf.nn.relu(layer_propagation)
layer_5_output = layer_5_activation

# Layer 6
weighted_input_l6 = tf.matmul(layer_5_output,
                              layer_6['weights'])
layer_6_propagation = tf.add(weighted_input_l6, layer_6['biases'])
layer_6_activation = tf.nn.relu(layer_6_propagation)
layer_6_output = layer_6_activation
```

Note that during network tuning described in subsection 4.6.6, a method that alters the output function is used.

Neuron and layer quantity

At the initial stage of the neural network construction phase, there was no concrete strategy for finding the optimal numbers of hidden layers and neurons. Therefore several networks with varying amounts of hidden layers and neurons were to be created, in order to evaluate and select the optimal layer set up.

Regardless of hidden layer and neuron quantity, all networks adhered to the same interconnection structure from input layer to the output layer. The process of altering the amount of hidden layers and neurons were quite painless due to layer creation being partially modular. The size of input and output layers remained the same throughout different networks.

Output layer for regression and classification

The model arrangement chosen for performing regression tasks consist of predicting one feature. In order for the network's output to be one single value, there is only one neuron in the output layer.

Classification tasks involve predicting which classes samples are a subset of. The method implemented to identify classes involved the network's outputs being as many as the amount of classes. Each output corresponds to the sample's probability to belong to a certain class.

Batch normalisation

The features of data used to train and evaluate models will have varying value ranges of which they can assume. This causes problems when features with small value ranges are neglected, in favour of features with large ranges, by the machine learning models. To prevent this, batches passed to the network will encounter a normalisation layer.

A simplified example of a batch normalisation implementation:

```
scale = tf.Variable(tf.ones([node_amount_input_layer]))
beta = tf.Variable(tf.zeros([node_amount_input_layer]))
batch_mean, batch_var = tf.nn.moments(input_layer,[0], name='moments')
batch_norm = tf.nn.batch_normalization(input_layer,batch_mean,batch_var
                                       ,beta,scale, 1e-3)

weighted_input = tf.matmul(batch_norm, layer_1['weights'])
layer_1_propagation = tf.add(weighted_input, layer_1['biases'])
layer_1_activation = tf.nn.relu(layer_propagation)
```

4.3.3 Selecting training components

The selection of training components can alter the models' learning rapidness and quality.

Performance measuring

The models constructed during the project uses cost functions instead of utility functions as performance measurements. This decision was made due to TensorFlow's optimisers computing gradients based on costs (synonymy called loss in reference)[9, Optimizers section].

Selecting a cost function that accurately depicts distances between predictions and desired outcomes can have a significant impact on the entire models' performance. Due to regression and classification tasks involving dissimilar desired outcomes, the regression and classification models need to have separate types of cost functions.

Cost functions for classification

Classification tasks involve predicting which class or classes a sample can be classified as. The cost function of a classification model needs to receive input from several neurons of the networks output layer and compare each neuron's emission to its desired value. The classification models created during this project trains on sets of samples, creating a need for having a cost function that would equitably determine the losses of all predictions made.

The cost for each sample needs to be calculated based on how similar its output is to its label. The function chosen to perform this was *softmax_cross_entropy_with_logits*, a built in TensorFlow operation that measures error probability in discrete classification. The function computes softmax cross entropy between predictions (in the form of log probabilities) and labels (in the form of a probability distribution).

The output of this function is a one dimensional tensor holding, one entry for every sample's calculated cost. In order to quantify the total cost of the tensor's content, the TensorFlow function *reduce_mean* is used to compute the mean of all elements of the tensor. (See Appendix, Figure B.6, for a simplified example illustrating the general methodology of this solution.)

Mean reducing the softmax cross entropy of all samples results in a good performance representation of the complete set of samples inserted. One drawback of using softmax cross entropy is that all classes are mutually exclusive. In essence, one sample must belong to exactly one class. The usage of another cost measuring strategy will be discussed in subsection 4.6.5.

Cost functions for regression

The regression task addressed in this project concerns the prediction of a single float value for each sample. The function chosen for measuring the total cost of a batch was root mean square error, full equation defined at section 2.7. (See Appendix, Figure B.7, for an example of a simple root mean squared error implementation.)

Optimisation algorithms

The initial optimiser for the models was TensorFlow's *GradientDescentOptimizer*, which is a optimiser that performs a gradient descent algorithm to minimise the cost of the network. The gradient descent optimiser resulted in satisfactory end result but converged slowly, requiring a lot of training steps to reach desirable results.

The gradient descent optimiser was only used during a brief period of the project, being replaced by the TensorFlow's *AdamOptimizer*. Implementing an adaptive moment estimation algorithm resulted in remarkably improved converge rates and costs for the models.

4.4 Session procedures

In order to perform TensorFlow operations, session objects needs to be created. A session encapsulate the environment of which all TensorFlow operations are executed in. The structure of two types of sessions, one for training and one for evaluation, will be presented and motivated in this section.

4.4.1 Training session procedures

In order to train the created models, a session containing training procedures needs to be declared.

Fundamental session components for training

The instructions for conducting the training are executed inside a session object. The instructions required to perform a single training step are:

- Initialising all global variables.
- Fetching the data to be used in the training step.
- Selecting the operations to be performed. (The session will automatically execute any operations that are needed for performing the selected operation.)
- Feeding the training data and other step specific variables to the placeholders.

The instructions for performing a single training step realised in code:

```
{Model placeholders, variables and operations are declared in forehand}
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    feature_set, labels = getNextBatch()
    sess.run([optimizer], feed_dict = {X: feature_set, Y: labels})
```

Scheduling batches and epochs

Preforming only one step of training would not give any useful result. Several thousands of training steps are needed to render fruitful results. A series of procedures were implemented for iterating through batches of training data. In order to train on all batches of data available, a loop for iterating through all batches is implemented. New batch data is fetched and fed to the placeholders at each iteration, along with running the training operations.

The instructions for performing several training steps on all batches, realised in code:

```
{Inside the session}
for batchnr in range(int(data_size/batch_size)):
    feature_set, labels = getNextBatch()
    {Run training operations}
```

Generally, training once on the complete set of training data is insufficient. The process of training on all batches needs to be repeated a number of times. An epoch is a complete training

on all available data set aside for training, equivalent to running the "batch loop" once. A loop for repeating the "batch loop" was implemented to run several epochs.

The instructions for performing several epochs, realised in code:

```
{Inside the session}
for epoch in range(training_epochs):
    for batchnr in range(int(data_size/batch_size)):
        feature_set, labels = getNextBatch()
        {Run training operations}
```

Providing feedback through the CLI

The Python program is executed inside a CLI, in which the program can print feedback to the user. Receiving useful information during training session, such as the number of epochs remaining and summed step costs for epochs, is a useful debugging feature that can indicate the training progress of the models. If unwanted model behaviour is observed, the training can be manually preempted saving time that would be spent on fruitless computation.

Print command added to the end of the epoch loop:

```
print("Epoch: ", epoch, " done out of: ", training_epochs,
      "\nEpoch cost: ", epoch_loss)
```

Saving the model

Training a model can take several hours, making it impractical having to retrain the model whenever the model is to be practically used. This was the reasoning for implementing training session preservation, making it possible to resume sessions for further evaluation or prediction.

To achieve this, the TensorFlow class *Saver* was implemented. *Saver* contains operations for saving and restoring model variables. The implementation was straightforward, creating a saver object inside the model and calling it inside the session when the model is to be saved.

Saving initialisation added to a session:

```
{Model placeholders, variables and operations are declared in forehand}

saver = tf.train.Saver()

with tf.Session() as sess:
    {Session initialised}
    {Training epochs are executed}
    saver.save(sess, save_path)
```

4.4.2 Evaluation session procedures

The models that have been successfully trained have presumably a acceptable capability to predict on samples from the training data. To test how well the model have generalised to its task, an evaluation session was created. The session introduces new data to the model and presents the prediction accuracy.

Restoring model variables

To restore the network and other model variables saved from the training session, an identical variable setup as the training session needs to be declared for the evaluation session. This was solved by using the same model as the training session.

The TensorFlow class *Saver* was used for restoring the variables saved prior evaluation.

The process of restoring sessions expressed in code:

```
{The same placeholders, variables and operations as training model
are declared in forehand}

saver = tf.train.Saver()

with tf.Session() as sess:
    tf.train.Saver().restore(sess, "save_path")
```

Defining accuracy operations

To better visualise the results emitted from the network, prediction and accuracy operations were defined. These operations are called inside the evaluation session and returns values quantifying the success of predicting test samples.

The accuracy operations for classification models using softmax cross entropy:

```
correct = tf.equal(tf.argmax(network_output, 1), Y)
accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
```

The *correct* operation takes the network output and compares it to the desired outputs, returning a list of Boolean data types having one "correct or incorrect" value for every sample tested. The accuracy function returns a percentage of how many samples was correctly predicted.

The accuracy operations for regression models:

```
difference = tf.abs(tf.subtract(Y, network_output)))
accuracy = tf.reduce_mean(difference)
```

The *difference* operation returns a list of absolute differences between each samples predicted output and desired output. The accuracy operation return the mean for all differences.

Evaluation procedure

The remaining step in evaluation is to fetch the training data and executing the operations needed for evaluating the models and printing the results.

A session for evaluating the restored model:

```
{The same placeholders, variables and operations
are declared in forehand}

with tf.Session() as sess:
    tf.train.Saver().restore(sess, "save_path")
    feature_set, labels = getTestData()
    diff, acc = sess.run([difference, accuracy],
                         feed_dict = {X: feature_set, Y: labels})

    print("Eval session accuracy: ", acc,
          "\nList of samples' differences:\n", diff)
```

4.5 Acquiring vehicle data

This section explains the process of gathering data and the structure of the acquired data.

4.5.1 RC car

The main data used in this project was gathered using a RC car. The RC car was provided, setup and operated by Benjamin Vedder. Vedder had both developed the code used to control the car and assembled the car itself.

The car was controlled by autopilot and used RTK (Real Time Kinematic) GPS in addition to other sensors to determine the direction and speed of the car. The routes to be driven were mapped out by manually driving the car and logging its positions. The car was equipped with an IMU (Internal Measurement Unit), containing an accelerometer, a gyroscope and a magnetometer. The IMU was a part of the car's controller board. It was also possible to log the current of the motor and both current and voltage of the battery. The rotational speed of the motor could also be measured with a tachometer.

4.5.2 Collecting the data

The RC car was driven on a parking lot with a relatively even surface. The routes had slight variations between runs. The courses was setup to include a few accelerations and deceleration as well as loose and sharp turns. This was done to hopefully better highlight the differences between the normal runs and the runs with faults that was later added. The courses were quite short due to the limited amount of space. They were repeated to achieve more log data.

Faults were introduced to the car with the criteria of being detectable by at least one of the car's sensors. The first fault served to simulate a possible suspension fault. This was done by loosening the suspension of a front wheel. For the second fault, a resistance was soldered on in series with the battery of the car. This was to emulate an electrical error, such as deterioration of wires or connectors.

4.5.3 Structure of the log files

The log files contained 28 types of values and were stored as CSV-files. (See Appendix A, Figure A.2, for a list of all of the value types.) Samples were logged 20 times a second and the resulting files contained approximately 30 000 samples. These were split into files without faults and separate files for each of the different faults.

4.6 Tuning the machine learning models

This section presents the different techniques that were utilised to fine tune and optimise the models.

4.6.1 TensorBoard

Tracking different model variables was seen as an useful ability for debugging purposes. To achieve this, the visualisation tool TensorBoard was implemented. To use TensorBoard, variable logging during TensorFlow session were needed to be configured. Sessions with logging activated generates log files readable by TensorBoard.

Running TensorBoard was near effortless, with the only configuration needed was to set the path of log file to be read. The plots drawn by TensorBoard was useful for detecting undesirable model behaviour. Figure 4.1 shows a TensorBoard plot rendering the amount of cost over training steps.

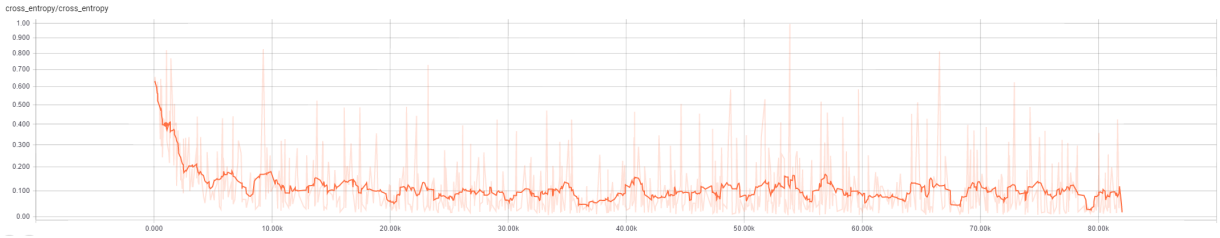

Figure 4.1: TensorBoard plotting a session with 1000 of epochs where x: training steps | y: cost

4.6.2 Evaluating activation functions

The choice of ReLU as activation function for both types of models was based on recommendations from literature and web blogs. The group did not know by own experience if there are other solutions that should be implemented in its stead. In order to distinguish ReLU as the optimal choice, several tests with different alternatives was performed.

Both regression and classification models performed indistinguishably when using ELU compared to ReLU. Knowing ELU involve more computation, due to logarithmic calculations on negative input, ELU was discarded as a potential replacement.

The classification models performed significantly worse when switching to TanH and logistic activation functions, converging much slower and reaching worse predictions. The regression model had about the same level of results as using ReLU. In addition the TanH and logistic activation functions are also more computation heavy than ReLU, making the activation functions not viable alternatives.

This evaluation process settled the choice of activation function for both models to remain ReLU.

4.6.3 Feature engineering

The Feature engineering process was an important step of the project. The collected vehicle data contained an abundance of features that could potentially interfere with the models' ability to predict, making feature selection a crucial step to carry out. Features involving GPS coordinates and timestamps was removed due to them potentially enabling cheating, E.g. the classification

model would detect class patterns based on position and time of testing.

In the case of classification, the model seemed to base its predictions heavily on the tachometer and calculated speed features. The consequence observed from this was incorrect pattern detection, making predictions in a peculiarly strange manner.

The predictions between the resistance and suspension fault classes were noticeably vague, making correct distinctions between the classes just 50 to 60 percent of the time. These features was removed at the last stages of the project where most model tuning was finished, but had a larger effect on the results than the all other tuning procedures. Training accuracy for all class predictions was increased from sub 60 percent to post 85 percent.

4.6.4 Initialisation methods for weights

An unwanted initialisation behaviour, in the way models converged, was observed in the CLI output printed by the models. During training, the model started to find a pattern in the data at immensely different rates. The number of epochs needed for reaching an accuracy over 50 percent could differ between 300 epochs and over 1000 epochs. This behaviour was examined and one potential cause was inadequate weight initiation.

All weights were initialised with normal distributed random values, with the values bound between the neuron amount of the previous layer and neuron amount of the current layer, using the TensorFlow function *truncated_normal*.

Another method for initialising weights called Xavier initialisation, designed to maintain the gradient scale the same throughout all sets of weights in the network, was tested. TensorFlow has a built-in function for performing Xavier initialisation named *xavier_initializer* in the *contrib.layers* module. [10] [1, p.277,278]

The final initialisation solution is as follows:

```
w_init = tf.contrib.layers.xavier_initializer()
layer_variables = {
    'weights':tf.get_variable(shape=[neuron_amount_previous_layer,
    neuron_amount_this_layer], initializer=w_init, dtype=tf.float32),
    'biases': .....{{unchanged}}..... }
```

Using this, the rate of which models started to find patterns was drastically improved. It decreased the amount of epochs needed to reach 50 percent training accuracy, to a range between 6 and 20 epochs.

4.6.5 Changing performance measurements

Sigmoid cross entropy for classification

The major drawback of softmax cross entropy is that it restricts the model to predict exactly one class for every sample. Another cross entropy function available in the TensorFlow library is *sigmoid_cross_entropy_with_logits*. This function calculates the sigmoid cross entropy based on predictions and labels, and do not restrict predictions to be of one class.

Sigmoid cross entropy also enables predictions to be of no class, which means that samples that are fault free can be defined by having no assigned class. This is considered a perk of sigmoid cross entropy due to enabling a more natural format for classes where only faults are marked as

classes, one output neuron represent the probability of one specific fault.

Mean absolute error

The predictions emitted from the regression model was of poor quality after initial construction. One potential fault of the regression model was the choice of cost function, root mean square error. Root mean square error respond harshly to large differences, exponentially increasing the cost impact of the error. This cost function was deemed poor for the models to use. A cost function that treats all differences equally would be more preferable.

Mean absolute error, defined in section 2.7, fulfil our new criteria for regression cost functions. Mean square error was implemented and tested to see if performance was improved:

```
diff = tf.abs(tf.subtract(desired_output, network_output))
cost = tf.reduce_mean(diff)
```

Sessions configured with very large amounts of epochs with different input batch sizes and activation function setups was performed, comparing the effectiveness between root mean square error versus mean absolute error. No real benefit was observed. The regression model had not improved its prediction capabilities in a beneficial way.

4.6.6 Reducing overfitting

The classification model made better generalisations than expected on the training data after extensive training sessions, making correct predictions a large majority of the time. However, the generalisation to the testing data was poor having the majority of its predictions incorrect. This phenomenon was quite evidently identified as overfitting. The model had gotten so complex that it was accounting noise and inconsistencies in the training data to its generalisation, instead of just the fundamental relationships [1, p.26]. The following subsections describes three methods that were evaluated to reduce overfitting.

Adding neuron-dropout to layers

Dropout is a method for reducing overfitting by randomly dropping connections between neurons. This restrains the neurons of a network from co-adapting excessively. [11]

On large network, with ten hidden layers or more which overfitted to an substantial degree, dropout significantly improved the accuracy of evaluations. The dropout did not have as desirable effect on smaller networks, such as the shrunk network solution in the next sub-subsection.

Reducing the amount of hidden layers

A more primitive way to reduce model complexity is to trim the amount of hidden layers in the network of the model.

By reducing the amount of hidden layers from ten to five layers, the testing accuracy went from sub 40 percent to past 60 percent in observed training sessions for classification. This strategy seemed to render the same results as dropout while reducing the amount of computation required at each training step, due to having a decreased amount of neuron activations to calculate.

Early quitting

Another seemingly primitive method for reducing complexity is early quitting. The number of epochs executed are reduced to a point where the model yet has not perceived the data noise and inconsistencies. Early quitting had a significant impact on testing performance on the models with large networks. The performance of the ten hidden layer model, mentioned in the sub-subsection above, was increased roughly in the same proportion as reducing the hidden layer amount to five.

But finding the amount of epochs optimal for testing were tedious. Sessions running identical models and configurations can have varying rates of which it starts to converge and ultimately overfitting, depending on weight and bias initialisation of the network. The varying points of which the model starts overfitting made it difficult to find the optimal point of which the model is moderately trained.

Using the early quitting method on a smaller and well balanced network resulted in underfitting, the model could not find any fruitful relationships before being preempted. The early quitting method was discarded in favour of reducing the amount of hidden layers due to the ease of configuration.

Result

5.1 Parser

Two parsers have successfully been constructed. The parsers have functions capable of converting CSV-files to data structures usable by the models. The parsers can also package training data in batches for model training and deliver testing data.

5.2 Classification model

The final classification model produced during this project is able to find patterns in the vehicle data and make competent predictions. The optimal model was configured with five hidden layers, sigmoid cross entropy cost function and no dropout. A representation of the model's accuracy can be examined in Figure 5.1.

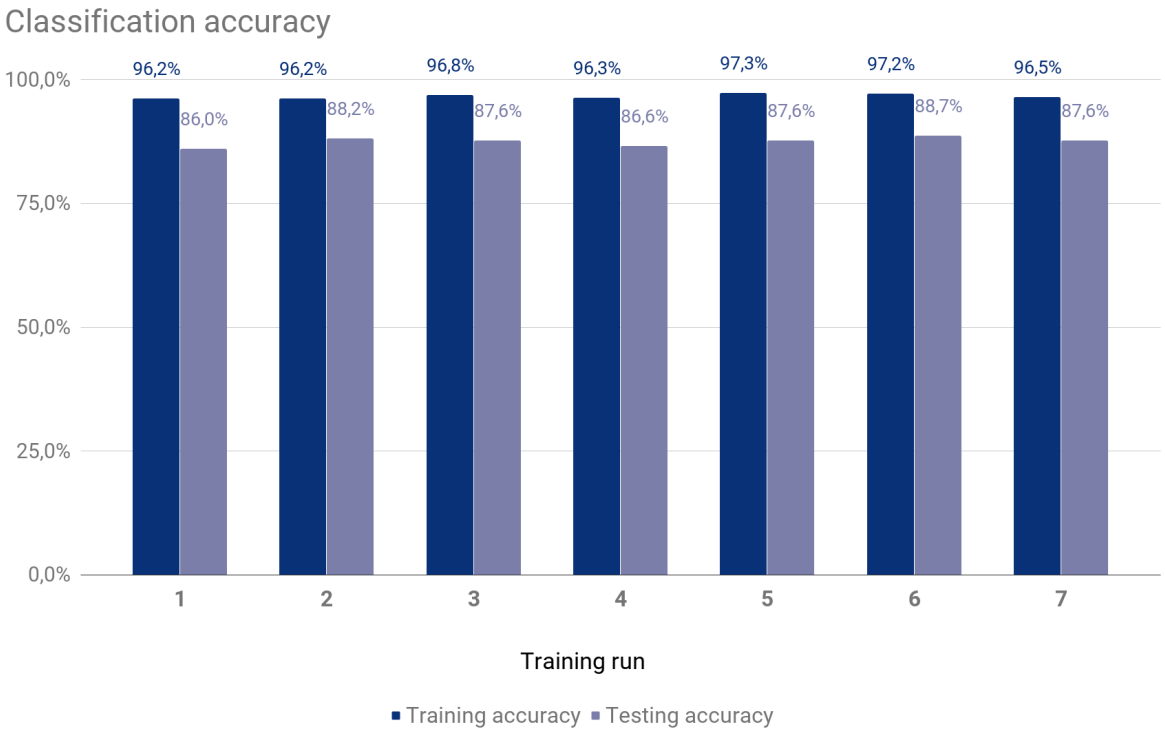

Figure 5.1: The classification model's prediction accuracy over 7 different training runs, each trained with 200 epochs. The training accuracy staples shows the model's prediction capabilities on the complete set of training data. The testing accuracy staples show the prediction accuracy on data reserved for evaluation, data which the model has not trained on.

The model render stable and accurate results throughout the training and evaluations. At no observed circumstances has the model's testing accuracy dipped under 85%.

Since the classification model predicts the status of each sample, it is relatively easy to find which samples are faulty. The percentage of samples deemed faulty can be used to analyse the likeliness of a vehicle having faults.

5.3 Regression model

The final regression model was not able to render any results adequate for analytic usage. The predictions made by the model resembled a mean value of all desired values instead of an active prediction for each sample.

Discussion and conclusion

This chapter discusses the project's results, model construction and the potential of using supervised machine learning for real world application.

6.1 Result assessment

The classification model could potentially preform better with further optimisation but the resulting accuracy from the model did reach an acceptable level. The model still has some generalisation issues with test data which was not completely resolved. This could be the reason behind the testing accuracy being a bit lower than the training accuracy. A higher training accuracy would be hard to reach due to some data being to similar for an accurate classification. An example of this is when the vehicle is stationary. The uncertainty of certain parts of the data could explain the variance in accuracy.

The regression model was not able to make any proper predictions regardless of layer and neuron counts. The variables estimated to be the most achievable as labels was the tachometer and motor current. They were seemingly depended on other variables, but not to a degree which our model could detect.

6.2 Potential real world application

A similar model to the classification model created in this project could work using data from a real world car. The created model is made for the data used in this project and would most likely not work optimally, or not at all, with different data. Real world data might improve the model's capability. Assuming real world cars' data contains a larger amount of sensor values usable as features, more faults could potentially be correctly classified. One issue might be that adding more types of faults would require substantially more data for each fault during training. Another issue would be that all faults needs to be manually labelled appropriately. This could potentially become easier if the data is first analysed with a regression model, to find which features differ from the expected value.

A regression model using real world data would not need as much preparation as the classification model. This is because the regression model does not require manual additions of labels nor any faulty data during training. A regression model could potentially be an aid for identifying the cause of faults in vehicles.

6.3 Using supervised machine learning for software life cycle management

The prediction capabilities demonstrated by the classification model shows that distinctions between faults can be made quite clearly. Real world implementations, with models more refined

than ours, should end up with similar or even better results in its ability to establish data patterns.

We believe that it is possible to use supervised machine learning as an helpful tool during the testing stages of a car's software life cycle. With the extended knowledge provided by supervised machine learning models, the vehicle industry could improve the management capabilities of the software development phases past testing. By having more information about the faults, better planning can be made to resolve issues.

Three examples of useful knowledge for software life cycle management that supervised learning could provide are:

- Using classification to find out if problems experienced resemble any of the faults the model has been trained on.
- Using regression to find how faults manifests throughout the features of a car. For example: Engine cooling malfunction leads to a specific set of features to increase notably.
- Using regression to find if and how faults happen in conjunction. For example: The battery outputs low voltage, which leads to several microcontrollers to malfunction.

6.4 Additional discussion

It was difficult for us to learn about machine learning and the tool TensorFlow, having no practical knowledge about machine learning at the start, during such a short period of time. Due to uncertainty about how much time was required for us to construct models, some choices in the project was rushed and not optimal. Correcting these missteps ended up taking more time than it would be to do them sufficiently in the first place.

Additionally there were small amounts of literature about TensorFlow and recent methods of machine learning. Even web forums like *Stackoverflow* lacked the answers for many questions we had during the project. The book *Hands-On Machine Learning with Scikit-Learn & TensorFlow* was a great step into machine learning and TensorFlow. It had a good content structure, pedagogical content for people unfamiliar with machine learning and was thorough describing various recommended machine learning methods.

Fortunately our mentors at Semcon was really helpful and supportive, giving us expert feedback on our work and proposing alternative methods for improving our results.

6.5 Critical discussion

Many development phases were rushed leaving some decisions not being thoroughly evaluated, leading to time spent on problem solving sub-optimal implementations. The missteps which cost us the most time and effort was:

- Overlooking features during feature selection.
- Not investigating enough on cost functions for regression.
- Using oversized networks for the models.
- Choosing poor weight initialisation methods.

The missteps had such a large impact that both models did not produce any results that even resembled correct predictions. In addition, the missteps listed were not discovered until the later stages of the project, costing us precious time that could have been spent on productive development. One solution which could have been better implemented, if its development phase was not rushed, was reduction of model overfitting. Further attempts to reduce model overfitting could have improved the evaluation accuracy of the classification model.

The labels used by the regression model did not have strong enough connections to other features. Making it hard to recreate the desired output. If better features was extracted from the test data, maybe the regression model could find a pattern to use when predicting its output.

6.6 Conclusion

The goals set up for the project has been completed:

- Two parsers have been made, customised for parsing and feeding data with regression labels and with classification labels.
- Supervised learning models have been created, and optimised to the extent we could, for the vehicle data available.
- The models' prediction ability have been evaluated.

By accomplishing the goals set up and examining the results, we achieved our purpose of investigating supervised learning as a tool for software life cycle management.

Supervised learning was reckoned to be a useful tool, for collecting more information, to be used when managing software life cycles.

Future work

A potential path to continue this project, would be to try to create a working regression model. This would most likely require collection of new data. This data should contain values that have a dependence on as many features as possible, values that would later be used as a label. An example of a possible label is the total power leaving the vehicle.

If a similar project would be done in the future then TensorFlow would be recommended to take into consideration. It allows for complex machine learning models to be created using higher level code, reducing both coding time and code length. This could be beneficial for less experienced developers, but become more of a hindrance for others. The data used is also important. A lack of data, or poorly selected features, could make predictions hard or even impossible to make. The possibilities of machine learning is limited by the data used.

References

[1] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, 2017.

[2] D. Kriesel, "A brief introduction to neural networks," Available at http://www.dkriesel.com/en/science/neural_networks, 2007, (accessed 7-June-2017). [Online].

[3] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*, ser. Springer Series in Statistics. Springer New York, 2009.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014, (accessed 9-June-2017). [Online]. Available: http://arxiv.org/abs/1412.6980

[5] Google Brain Team, "Tensorflow api documentation," www.tensorflow.org/api_docs, (accessed 8-June-2017). [Online].

[6] Nvidia Corporation, "NVIDIA CUDA® Deep Neural Network library," https://developer.nvidia.com/cudnn, (accessed 7-June-2017). [Online].

[7] The MathWorks Inc., "Parallel hybrid transmission," www.se.mathworks.com/help/physmod/sdl/examples/parallel-hybrid-transmission.html, (accessed 9-June-2017). [Online].

[8] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *CoRR*, vol. abs/1609.04836, 2016, (accessed 9-June-2017). [Online]. Available: http://arxiv.org/abs/1609.04836

[9] Google Brain Team, "Tensorflow api documentation," www.tensorflow.org/api_guides/python/train, (accessed 8-June-2017). [Online].

[10] Google Brain Team, "Tensorflow api documentation, initialisers," www.tensorflow.org/versions/r0.11/api_docs/python/contrib.layers/initializers, (accessed 8-June-2017). [Online].

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014, (accessed 10-June-2017). [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

Appendix A

Cifar 10 image recognition training performance:
CPU: Intel i7-4770k @stock | GPU: NVIDIA 980ti @stock
Oscar's Workstation(CUDA) : $1300 - 1600 samples/sec$
Oscar's Workstation(CPU) : $240 - 260 samples/sec$

Figure A.1: These are benchmarks demonstrating the power of GPU-acceleration

Value types in log data	
Variable name	**Description**
time	Timestamp
temp_mos	Temperature
current_in	Battery current
current_motor	Motor current
v_in	Battery voltage
px	Car x position
py	Car y position
lx	GPS x position
ly	GPS y position
lz	GPS z position
ix	Initial GPS ENU x
iy	Initial GPS ENU y
iz	Initial GPS ENU z
speed	Speed
roll	Roll
pitch	Pitch
yaw	Yaw
accel[0]	Accelerometer1
accel[1]	Accelerometer2
accel[2]	Accelerometer3
gyro[0]	Gyroscope1
gyro[1]	Gyroscope2
gyro[2]	Gyroscope3
mag[0]	Magnetometer1
mag[1]	Magnetometer2
mag[2]	Magnetometer3
tachometer	Tachometer
steering_angle	Commanded steering angle

Figure A.2: List of value types in the log data of the RC car

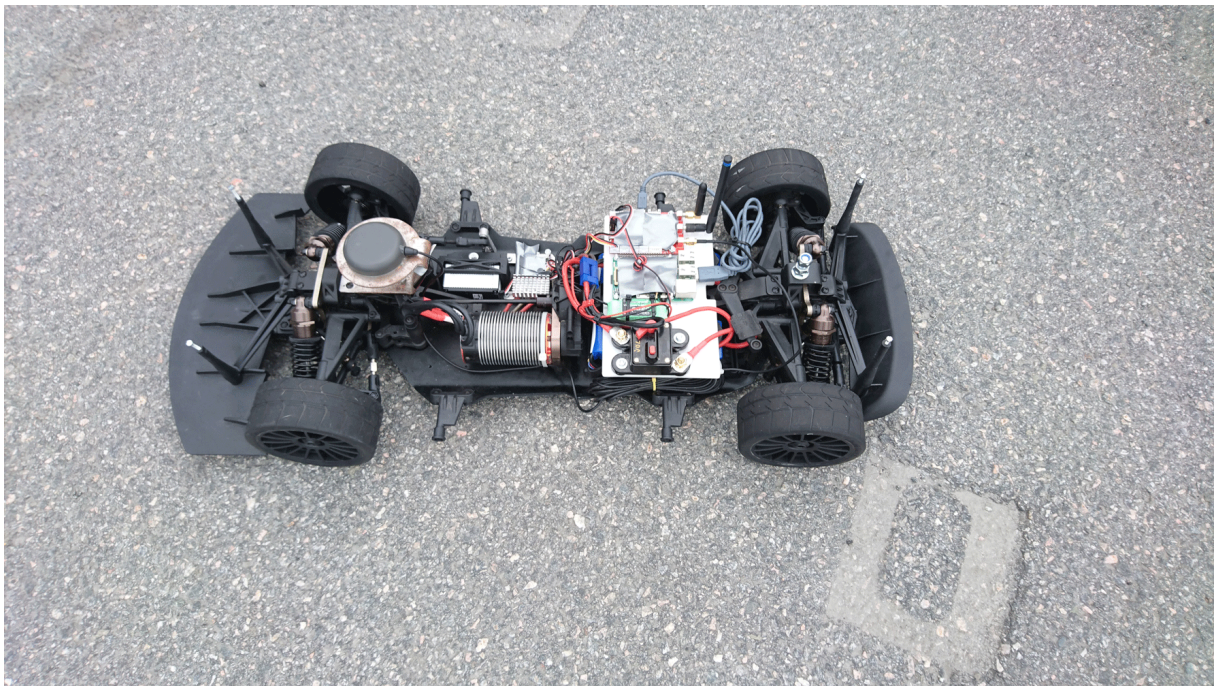

Figure A.3: Picture of the RC car.

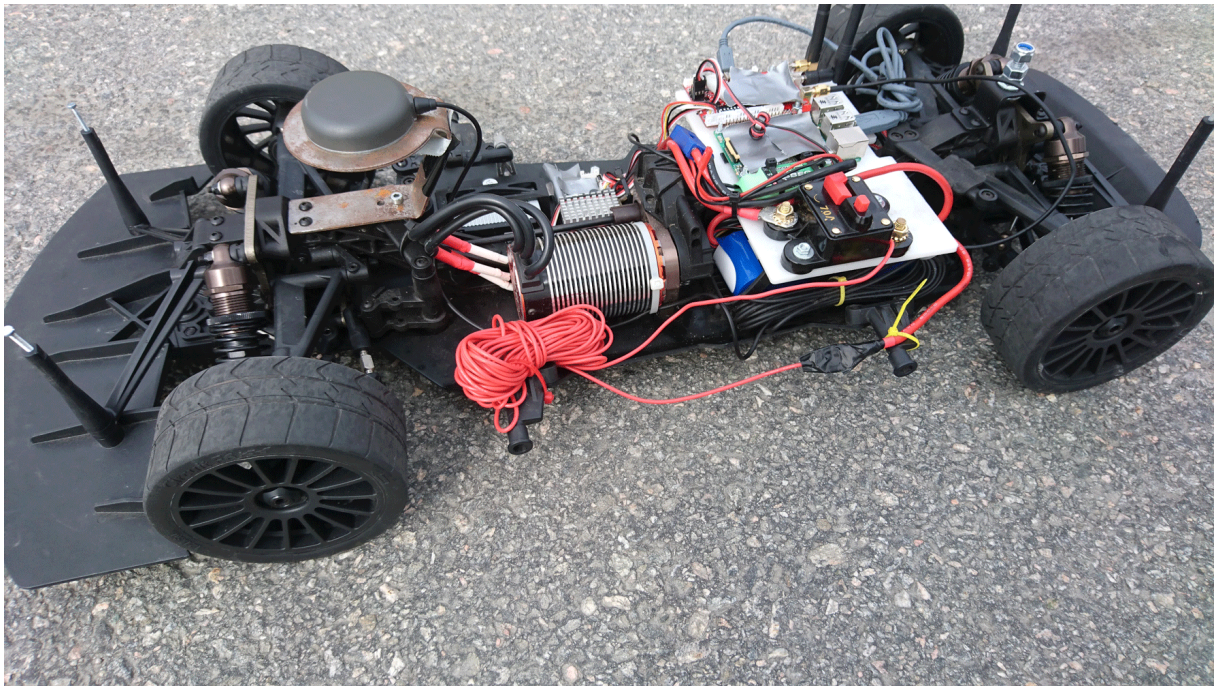

Figure A.4: Picture of the RC car with resistance added.

Appendix B

This appendix contains additional code

```
array = 2d-array
for row in CSVfile:
    {split the row of comma separated values to a list}
    {append the list to the array}
```

Figure B.1: This is psedu code for parsing CSV-files containing machine learning data.

```
newArray = numpy.delete(oldArray, [unwanted columns], axis=1)

#  "unwanted columns" is an array of int marking columns to be removed
#  "axis" marks which axis to remove elements along
#   axis is set to 1 for column based removal
```

Figure B.2: Demonstration of numpy.delete function.

```
newArray = []
for sample in range_of_oldArray:
  new_sample = []
  for delta in range(deltaAmount):
    {join time deltas to new_sample}
  {append new_sample to newArray}
```

```
Example of adding one time delta:
     Before              After
    [[a,b,c],          [[d,e,f,a,b,c],
     [d,e,f],           [g,h,i,d,e,f],
     [g,h,i],           [j,k,l,g,h,i]]
     [j,k,l]]
```

Figure B.3: A function that adds so called "time deltas" to an array.

```
def getNextBatch():
    global pointer1  # Pointer for data of type 1
    global pointer2  # Pointer for data of type 2
    global pointer3  # Pointer for data of type 3

    X, Y = arrays acting as placeholders batch data.

    for _ in range(batchLoop):
        pointer1 = pointer1 % (train_array1_len-sectionSize)
        X1 = getData(train_data1, pointer1, pointer1+sectionSize)
        Y1 = getData(train_labels_1, pointer1, pointer1+sectionSize)
        pointer1 += sectionSize

        X = numpy.vstack((X, X1))
        Y = numpy.vstack((Y, Y1))

        pointer2 = pointer2 % (train_array2_len-sectionSize)
        X1 = getData(train_data2, pointer2, pointer2+sectionSize)
        Y1 = getData(train_labels_2, pointer2, pointer2+sectionSize)
        pointer2 += sectionSize

        X = numpy.vstack((X, X1))
        Y = numpy.vstack((Y, Y1))


        pointer3 = pointer3 % (train_array3_len-sectionSize)
        X1 = getData(train_data3, pointer3, pointer3+sectionSize)
        Y1 = getData(train_labels_3, pointer3, pointer3+sectionSize)
        pointer3 += sectionSize

        X = numpy.vstack((X, X1))
        Y = numpy.vstack((Y, Y1))

    return X, Y
```

Figure B.4: This is a batch fetcher for classification models with 3 sets of data.

```
def getNextBatch():
    global pointer1  # Pointer for data of type 1

    pointer1 = pointer1 % (train_array1_len-batchSize)
    X = getData(train_data1, pointer1, pointer1+batchSize)
    Y = getData(train_labels_1, pointer1, pointer1+batchSize)
    pointer1 += batchSize

    return X, Y
```

Figure B.5: This is a batch fetcher for regression models, only one set of data is used.

```
desired_output = [ [1,0,0],  # Sample 1
                   [0,1,0],  # Sample 2
                   [0,0,1],  # Sample 3
                   [0,1,0],  # Sample 4
                   [1,0,0]]  # Sample 5

network_output = [ [1.5, 0.1, 0.3],  # Sample 1, good prediction
                   [0.1, 2.0, 0.4],  # Sample 2, good prediction
                   [0.2, 1.1, 0.3],  # Sample 3, poor prediction
                   [0.1, 0.3, 2.5],  # Sample 4, really poor prediction
                   [3.0, 0.5, 0.3]]  # Sample 5, really good prediction


{Inside a tf.Session object}
diff = tf.nn.softmax_cross_entropy_with_logits(logits=network_output,
                                               labels=desired_output)
cost = tf.reduce_mean(diff)


print(diff) --> [ 0.43682885   # Sample 1, low cost
                  0.30118927   # Sample 2, low cost
                  1.41836905   # Sample 3, high cost
                  2.38358831   # Sample 4, really high cost
                  0.13914485]] # Sample 5, really low cost

print(cost) --> 0.935824
```

Figure B.6: A demonstration of one classification model's cost function, predicting on 3 possible classes and a batch of 5 samples has been passed through the network.

```
desired_output = [ 10.0,  # Sample 1
                   20.0,  # Sample 2
                   30.0,  # Sample 3
                   20.0,  # Sample 4
                   10.0]  # Sample 5

network_output = [ 11.0,  # Sample 1
                   18.0,  # Sample 2
                   33.0,  # Sample 3
                   20.0,  # Sample 4
                   3.0]   # Sample 5

{Inside a tf.Session object}
diff = tf.square(tf.subtract(desired_output, network_output))
cost = tf.sqrt(tf.reduce_mean(diff))

print(diff) -->  [ 1.0   # Sample 1, low cost
                   4.0   # Sample 2, mediocre cost
                   9.0   # Sample 3, high cost
                   0.0   # Sample 4, really low cost
                   49.0] # Sample 5, really high cost

print(cost) --> 3.54965
```

Figure B.7: A demonstration of a root mean squared error function, predicting 1 variable and a batch of 5 samples has been passed through the network.