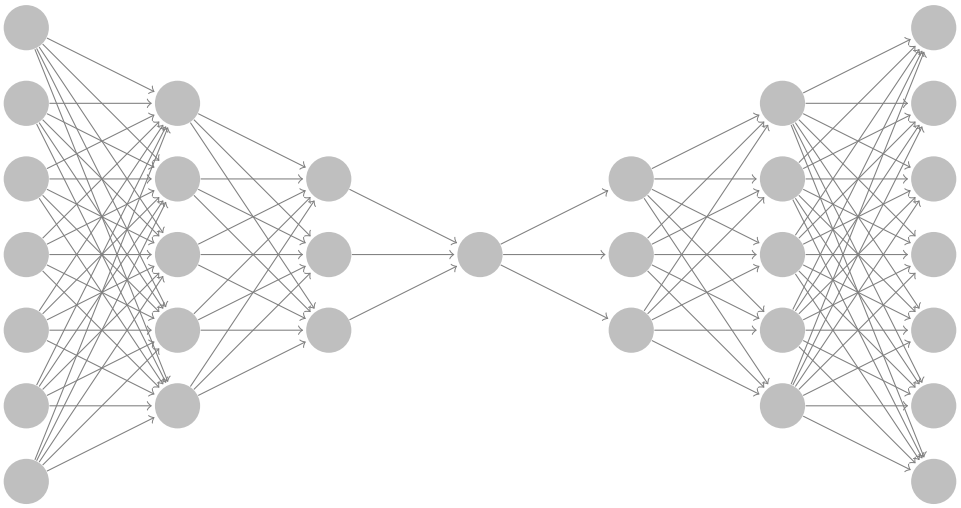
CHALMERS
UNIVERSITY OF TECHNOLOGY



Software Lifecycle Management

Unsupervised Anomaly Detection

Ludwig Friborg
Victor Christoffersson

Examinator: Peter Lundin

Institutionen för data- och informationsteknik

CHALMERS TEKNISKA HÖGSKOLA

GÖTEBORGS UNIVERSITET

Göteborg, Sverige 2017

This page has been intentionally left blank

Software Lifecycle Management
Unsupervised Anomaly Detection

Ludwig Friborg
Victor Christoffersson

Spring 2017

Examinator: Peter Lundin
Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Gothenburg, Sweden 2017

Software Lifecycle Management with Unsupervised Anomaly Detection
LUDWIG FRIBORG
VICTOR CHRISTOFFERSSON

Examinator: Peter Lundin
Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet
SE-412 96 Göteborg
Sverige
Telefon: +46 (0)31-772 1000

Cover: Illustration of a artifical neural network belonging to a autoencoder
Göteborg, Sverige 2017

Abstract

The purpose of this thesis is to evaluate if unsupervised anomaly detection, the task of finding anomalies in unlabelled data, can be used as a supportive tool for software life cycle management in finding errors which are tedious to detect manually.

The goal is to apply the techniques of unsupervised machine learning on data-sets that are collected and analysed from a miniature-scaled research vehicle system that resembles the operation of a real automotive vehicles electrical architecture.

Using a stacked autoencoder implemented with TensorFlow, the final application is able to detect anomalies within the collected data-sets from the research vehicle. This proves the concept of utilising machine learning for error detection as a viable method. Finally concluding whether the techniques of unsupervised anomaly detection is applicable on a larger scale for real automotive vehicles.

Keywords: Machine learning, Unspervised Anomaly Detection, Autoencoder, Software Life-cycle Management, Artificial Nural Networks.

Samanfattning

Syftet med denna rapport är att utvärdera om maskininlärning med tekniken *unsupervised anomaly detection* går att använda som ett stödjande verktyg för felsökning bland omärkt data. Den omärkta datan är insamlad från ett forskningsfordon i miniatyrskala vars syfte är att efterlikna driften av ett riktigt fordons elektriska arkitektur.

Den slutliga applikationen använder sig av en *stacked autoencoder* implementerad med hjälp av ramverket TensorFlow. Applikationen kan finna anomalier bland den insamlade datan från forskningsfordonet och genom att göra detta bevisas konceptet av att använda maskininlärning med *unsupervised anomaly detection* som metod för fel-detektering. Tekniken kan möjligen tillämplas i en större skala för riktiga fordon.

Nyckelord: Maskininlärning, Unsupervised Anomaly Detection, Autoencoder, Artificiella Neurala Nätverk.

Preface

We went into this project with little to no prior knowledge within the field of machine learning and because of this we found it impressively how easy and quick it is to grasp the concepts of machine learning with TensorFlow. Though at this time (2017) the amount of resources is scarce, we would like to mention the book *Hands-On Machine Learning with Scikit-Learn & TensorFlow*[1] by Aurelien Geron which provided a great introduction and understanding of the fundamentals of machine learning and TensorFlow[7].

We would like to thank our mentors from both Chalmers and Semcon, Christer Ek, Peter Nordin and Christer Carlsson for providing us with insights and feed-back in planning and delivering this project. We would also like to thank Benjamin Vedder helping us gathering data.

Terminology

Bias - Bias is the value which determines how easily the activation function of the neuron is triggered. The lower the bias the harder it is for the neuron to activate.

Convergence - The point at which the artificial neural network reaches a good solution which is able to generalise to new data.

Hidden layers - A hidden layer is a layer which does not interact with the surroundings of the neural network besides the input and output of the neural network. Because these layers reside between input and output they are referred to as hidden layers.

Hyperparameter - Parameters used for calibrating the machine learning process.

Layers - A layer is either a input/output layer or a layer of neurons. Input and output layers are usually not called layers and are respectively called input and output of the neural network. Layers of neurons between the input and output are however called layers.

Machine Learning - A term describing the concept of a system coming to conclusion based on past experience.

Model - In context of TensorFlow the term model usually describes a trained network, where its weights and biases already are calibrated.

Neuron - A neuron consists of the weighted sum passed into an activation function which determines the neurons decision. There exists several different activation functions for neurons which makes decisions in different ways. Neurons acts as decision makers.

Supervised/Unsupervised learning - Whether the process is supervised or not is determined if there is a given training set of data.

TensorFlow - TensorFlow is a framework developed by Google dedicated for machine learning.

Weight - The weight is a general term to describe the importance of an input into a neuron. It is usually a floating point value.

Weighted sum - The weighted sum is the sum of all weighted inputs into the neuron and the neurons bias.

Contents

1. Introduction

1.1 Background

The work of this thesis is conducted at Semcon and is a part of project NGEA "Next Generation Electrical Architecture". The NGEA project in its own turn is part of the larger research project FFI "Fordonsstrategisk Forskning och Innovation" [9] between Vinnova and the automotive industry which aims to promote growth by supporting development of transportation technology and the Swedish automotive industry.

With automotive vehicles becoming more complex at an increasingly rapid pace, the need to monitor and troubleshoot these vehicles systems grows in parallel. These systems consist of advanced embedded electrical systems which together form an intricate network of communicating devices. The massive amount of messages sent over the electrical networks used in automotive vehicles during intercommunication can therefor create an problem since the origin of errors and exceptions can become obscure.

Detecting the origin of such errors can thus become time-consuming and costly to repair. A increasing need for supportive tools has become apparent. Recently the use of machine learning has become more common in the field of computer science and is becoming more incorporated into the development process of complex systems such as modern vehicles. Machine learning might therefore provide a solution to the issue of recognising patterns in data and pointing out anomalies.

1.2 Purpose

The purpose of this thesis is to research and evaluate if using unsupervised anomaly detection is applicable to detect unusual behaviour in the electrical architecture of automotive vehicles.

The aim is to conduct machine learning and apply techniques for unsupervised anomaly detection and evaluate this network using measurement data collected throughout the duration of the thesis. The network will either be deemed to work correctly if it is able to find artificially induced errors in the measurement data. The results of the network will then be used to evaluate if the used techniques is applicable or not for detecting unusual behaviour using unsupervised anomaly detection.

Furthermore this report will also describe different implementations of machine learning that could be used to achieve a desirable artificial neural network.

1.3 Method

To achieve the purpose of the project an application will be developed written in python. The application will be able to train a artificial neural network with measurement data. It will also be able to test data to find anomalies. The measurement data used with the application will be collected from a model vehicle run by Benjamin Vedder [8]. Multiple measurements will be sampled, some with no fault and some with induced anomalies. The measurements with induced anomalies include two separate runs, one with faulty suspension and a second one with a faulty resistance.

1.4 Limitations

The project will use the framework TensorFlow for construction of the neural network. To perform anomaly detection the application will use an stacked-autoencoder. The training-data will not be measurement data from a real automotive vehicle's electrical architecture but instead it will come from a miniature-scaled research vehicle.

2. Technical background

2.1 What is machine learning

In 1959 Arthur Samuel explained machine learning as the "field of study that gives computers the ability to learn without being explicitly programmed." [4, p. 1].

Machine learning is the science of using training data-sets to learn computer programs certain patterns. These patterns can be used to solve and optimise problems that otherwise would have been difficult to solve. Machine learning has been around for decades. From the research of Frank Rosenblatt's research about the Perceptron binary classifier during the late 1950s to moderns frameworks like Google's TensorFlow which powers utilities such as image and speech recognition.

2.2 Components of machine learning

The main components of a machine learning system is the data-sets that the system will learn from as well as the system itself. Hence these two components are the main challenges of creating an effective learning system. The data-sets can be "bad data" and the system can implement a "bad learning algorithm". [1, p. 22]

The common approach to creating these kind of learning systems is by using artificial neurons inspired by their biological counterpart the neuron of the human brain. By connecting several artificial neurons together an artificial neural network is created. Just as toddlers learn what an apple is by being shown an apple sufficient amount of times an artificial neural network works in a similar way, automatically inferring rules given enough examples, the training data-sets. [5]

2.2.1 Neurons as activation functions

An artificial neuron is simply a model which given inputs computes if its output should be fired or not. The most simple type of neuron would be the perceptron neuron which outputs either true or false given enough of its inputs are active, a binary classifier. While other neurons can output clamped values such as the Sigmoid neuron which can output a value between zero and one depending on its given inputs.

What the neurons outputs given their respective inputs are dependant on the *activation function* σ. Thus the choice of activation function determines what type of neuron it is. The input to the activation functions is generally called the *weighted sum + bias*, henceforth referred to as z, and is a summarised measure of how important each input is to its neuron firing or not.

The formula for a neurons output is thus: $output = \sigma(z)$

We now have the fundamentals to create an artificial neural network. To learn the network the weights are modified slightly and the difference of the output is observed. This way a difference in z, $\Delta(z)$ will be relative to a difference in output $\Delta(\sigma(z))$. This is relevant when developing artificial neural networks because a change in input, $z + \Delta(z)$ which pushes the output, $\sigma(z) + \Delta(\sigma(z))$, closer to a desired output which is effectively a change of a model-feature of the network. [5]

The perceptron neuron

In Michael Nielsens book Neural Networks and Deep Learning we find the concept of perceptrons which is the most basic type of artificial neuron. The perceptron takes several binary inputs, $x_1, x_2, ..., x_n$ and produces a single binary output, see *figure 2.1.* We will refer to the binary inputs collectively as the input vector $\hat{x}$ henceforth. The binary inputs in $\hat{x}$ each has a corresponding weight in whats known as the neurons weight vector $\hat{w} = [w_1, w_2, ..., w_n]$ which are real numbers expressing the importance of their corresponding binary inputs. The neuron's output is determined by whether the *weighted sum*, $ws = \sum_j w_j x_j$ is above some set threshold value. The threshold is a real value parameter of the neuron just like the weights[5]. This can be expressed with the following formula:

$$\sigma(ws) = \left\{ \begin{array}{l} 0 \text{ if } \sum_j w_j x_j \leq threshold \\ 1 \text{ if } \sum_j w_j x_j > threshold \end{array} \right\}$$

One can think of the perceptron as making decisions by weighing up evidence of varying importance. The above equation is however a bit complex and can be re-written with $ws = \sum_j w_j x_j$ as the dot product between $\hat{w} \cdot \hat{x}$ and moving the threshold to the other side of the inequality, and to replace it by what's known as the perceptrons bias, $b = -threshold$. With the above changes we can rewrite the formula for the perceptrons output as the following, where z is defined as the *weighted sum* + the *bias*: $\hat{w} \cdot \hat{x} + b$ [5]. This can be expressed with the following formula:

$$\sigma(z) = \left\{ \begin{array}{ll} 0 & \text{if } \hat{w} \cdot \hat{x} + b \leq 0 \\ 1 & \text{if } \hat{w} \cdot \hat{x} + b > 0 \end{array} \right\}$$

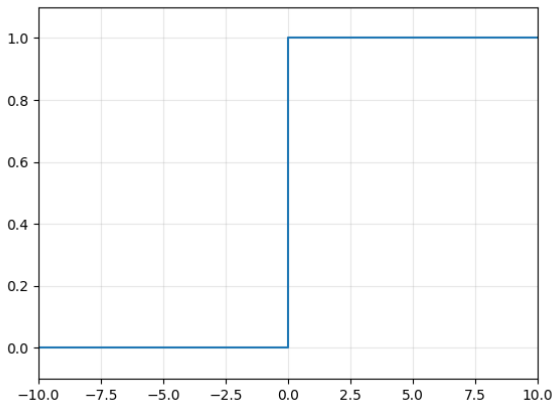


Figure 2.1: Graph of the Perceptron activation function.

When learning an artificial neural network using perceptrons a slight change in weights can cause the perceptron to swap status from 0 to 1. This behaviour can be far too drastic and may cause the rest of the network to misbehave. [5]

The logistic neuron

The logistic neuron, even known as the Sigmoid neuron outputs a clamped real value between 0 and 1 instead of binary outputs. One advantage over a perceptron is that the logistic function can estimate probability [1, p. 134]. When learning a network using the logistic function it is a huge advantage over using the percepton since a slight change of weights will only trigger a slight change in it's output [5]. This makes networks using the logistic activation functions much more well behaved during learning than perceptrons. The *logistic* activation functions definition takes the input z which is defined as the *weighted sum* + the *bias*: $\hat{w} \cdot \hat{x} + b$. This formula can be expressed with the following formula:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The *logistic* activation functions graph is seen in *figure 2.2*.

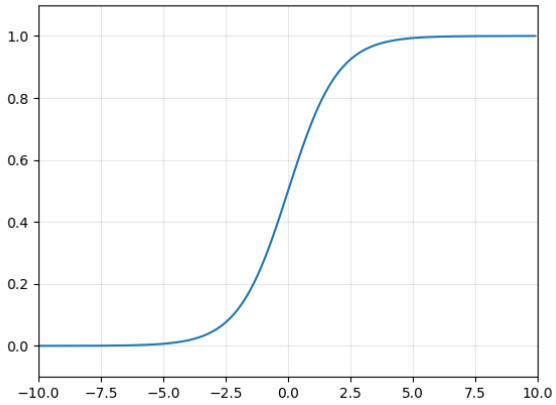


Figure 2.2: Graph of logistic activation function.

The rectified linear neuron

The linear rectifier neuron, commonly referred to as $ReLU$ is a linear function that returns the maximum of 0 and it's input. In this sense the $ReLU$ is continuous as long as it's input is positive [1, p. 279]. The $ReLU$ activation functions definition is seen below , where z is defined as the *weighted sum* + the *bias*: $\hat{w} \cdot \hat{x} + b$:

$$\sigma(z) = \left\{ \begin{array}{ll} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{array} \right\}$$

$ReLU$ might provide a more responsive and accurate model but it also puts greater requirements on the input, the neuron has a zero-gradient for $z < 0$ and will possibly reset weights and make the network forget what previously had been learned. This issue is called the *dying unit issue* [1, p. 279]. The $ReLU$ activation functions graph is seen in *figure 2.3*.

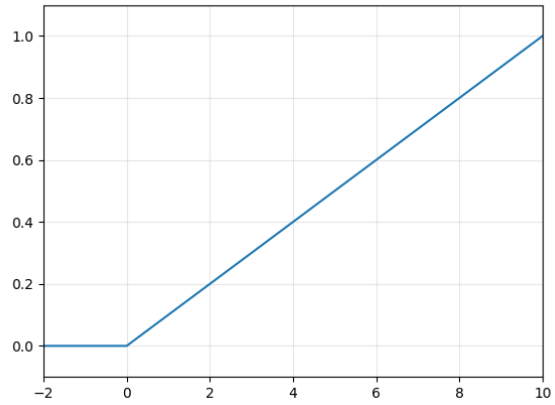


Figure 2.3: Graph of the ReLU activation function.

The $ReLU$ is continuous but not differential because its output abruptly changes from zero to linear when $z > 0$. This will just like the perceptron have an effect during learning since a slight change in weights can abruptly change the output. However $ReLU$ is generally very fast to compute and doesn't have a clamped output value like the logistic function. [1, p. 279]

The exponential linear neuron

The exponential linear unit ELU activation function is a version of the $ReLU$ with the difference that it takes on negative values when $z < 0$, acting like the logistic neuron discussed earlier. For input values $z > 0$ the function acts like the $ReLU$ neuron and outputs a linear output. This gives the ELU activation function a smoother transition output when the input values shift between positive and negative and is differential [1, p. 279]. The ELU activation functions definition is seen below and its graph is seen in *figure 2.4*, where z is defined as the *weighted sum* + the *bias*: $\hat{w} \cdot \hat{x} + b$:

$$ELU(z) = \left\{ \begin{array}{rl} e^z - 1 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{array} \right\}$$

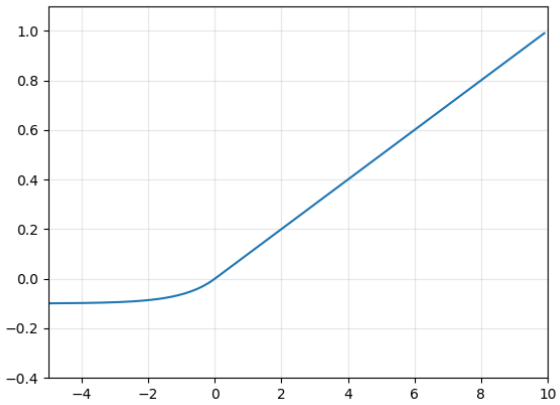


Figure 2.4: Graph of the ELU activation function.

The main advantages of the *ELU* function is that it takes on negative values when $z < 0$ this alleviates the *vanishing gradients problem*, see section 2.4.2. It also has a nonzero gradient when $z < 0$ which helps neurons which has a negative input from dying, avoiding the *dying unit issue*. [1, p. 281]

2.2.2 Deep neural networks

When a network has several layers of neurons the network is called a *Deep neural network*. A neuron layer in a feed-forward network is any number or neurons which aren't connected to each other in the same layer but to any given neurons in the upcoming layer, as seen in *figure 2.5*. The input and output to the network aren't neuron layers since they just pass on values. All neurons layers between the input- and output-layer are called hidden-layers. These multi-layered network architectures have the advantage of being able to learn more complex structures than single-layered networks. [1, p. 275]

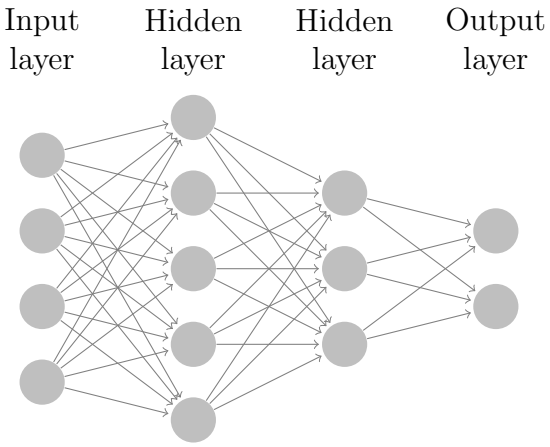


Figure 2.5: Graph of a multi-layered artificial neural network.

Backpropogation of error

As mentioned in 2.2.1 the fundamentals in learning artificial neural networks is to establish if a slight change of either *weights* or *biases* or both pushes the output towards a desired output, this difference between actual output and desired output is commonly referred to as the networks *output error* [1, p. 261]. Backpropogation of error works by feeding an input to the network and computes the output for each neuron in each consecutive layer eventually leading up to an output. The networks *output error* is measured and then the algorithm goes backwards measuring how much each neuron in each layer contributed to the *output error*, this measure is called the *error-gradient*. [1, p. 261]

Training neural network using optimisers

The actual training step takes the error-gradient and slightly changes the weights and neurons for each consecutive layer in proportion to how much each neuron contributed to the *output error* [1, p. 262].

While training the neural networks different types of units can be used, called optimisers. Optimisers utilises algorithms to adjusts weights and biases in the networks neurons. There are

plenty of different algorithms that can be used to determine which and how much the weights and biases should be adjusted. [1, p. 293]

The loss function

The measure of the *output error* of the network can be measured in different ways. A loss function is commonly implemented which measured the desired output. [1, p. 20]

In this thesis three loss functions will be implemented, the Mean Square Error(MSE), Root Mean Square Error(RMSE) and Mean Absolute Error (MAE) [1, p. 37]. These loss functions can be seen below:

$$Loss_{(RMSE)} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\hat{out} - \hat{in})^2}$$

$$Loss_{(MSE)} = \frac{1}{m}\sum_{i=1}^{m}(\hat{out} - \hat{in})^2$$

$$Loss_{(MAE)} = \frac{1}{m}\sum_{i=1}^{m}\left|\hat{out} - \hat{in}\right|$$

Where $\hat{out}$ and $\hat{in}$ is the output and input vector of the network. [1, p. 37]

The RMSE is defined as the square root over the average squared error while the MSE is the average squared error. The MSE is therefore faster to compute because the square root is a costly operation of computing power [1, p. 37]. The third method, MAE provides a performance advantage similar to MSE but measures the average error orthogonal. [1, p. 39].

Gradient descent optimisation

Gradient descent uses incremental steps to optimise the networks learning, slowly trying to adjust the weights and biases by using the error-gradient*, see backpropogation of error.* The parameter *learning rate* usually determines the size of adjustment. A common issue with gradient decent is plateaus where the algorithm gets stuck, not learning anything new. Gradient descent is fairly generic and can be used to optimise a vast variety of networks though high performance might not be guaranteed. [1, p. 111]

Momentum optimisation

Performing optimisation with Momentum involves the task of smoothing out the learning curve, not acting on unexpected anomalies in the training data-set. Momentum optimisation takes the features history for account and adapts to changes in the pattern slowly which allows the optimiser to easier escape getting stuck in plateaus. In deep learning Momentum is usually better than *gradient descent* at optimisation. [1, p. 294]

Adam optimisation

Adam stands for *adaptive moment estimation.* It keeps track of an exponentially decaying average of past error-gradients as well as the exponentially decaying average of past squared error-gradients. [1, p. 298]

Training with mini-batches

When training with mini-batches the *gradient* is computed over small sets of samples called *mini-batches* [1, p. 15]. Instead of feeding the network a single sample, $\hat{x} = [x_1, x_2, ..., x_n]$ with *mini-batches* the input to the network is a two-dimensional vector of input-vectors:

$$\hat{x} = [[x_1, x_2, ..., x_n], [x_1, x_2, ..., x_n], [x_1, x_2, ..., x_n]]$$

Mini-batch training has the advantages that when the mini-batches are large enough the loss optimisation becomes less erratic and converges better. A disadvantage is that it might have a hard time escaping a local loss minimum, missing the global minimum. [1, p. 15]

2.3 Unsupervised anomaly detection

Artificial neural networks utilising unsupervised learning has the property of not needing to know how an anomaly looks prior to training which makes it more versatile compared to the alternative of supervised learning [2]. The concept of unsupervised anomaly detection is to find patterns within the training data-set that defines normal behaviour. By learning a generalisation the network can then tell whether a new sample likely is an anomaly or not in the test data-sets. [1, p. 12]

2.3.1 Stacked autoencoder

A stacked autoencoder is a deep neural network which has the shape similar of a funnel, it consists of several hidden-layers with each hidden-layer containing consecutive less neurons than the previous towards the centre. Past the centre the networks expands the hidden layers in a similar fashion with each hidden layer containing increasingly more neurons than the previous layer. This can be seen in *figure 2.6*. The purpose of an autoencoder is to learn efficient representation of the input data-set, recognising a pattern within the set.

This is achieved by putting constraints on the network while trying to reconstruct the autoencoders input as the produced output. Since the hidden layers of the autoencoder decreases in size towards the centre the input data is compressed and the deep neural network effectively has to learn an efficient representation of the input data. Depending on how well the autoencoder is able to reconstruct the given input it is possible to tell whether the input follows the pattern of the trained data. [1, p. 417]

To do this the autoencoder consists of two necessary parts, the *encoder* which converts the inputs to an internal efficient representation. And the *decoder* which converts the internal representation to the outputs [1, p. 414]. In *figure 2.6* below the *encoder* consists of first four hidden layers where each column of dots represents a hidden layer of an arbitrary number of neurons. The hidden layer in the centre is the internal representation and the *decoder* consists of the four last hidden layers. As seen in *figure 2.6* the autoencoder has the same number (same dimensionality) of neurons for the input as the output, this is important since the output is a reconstruction of the input.

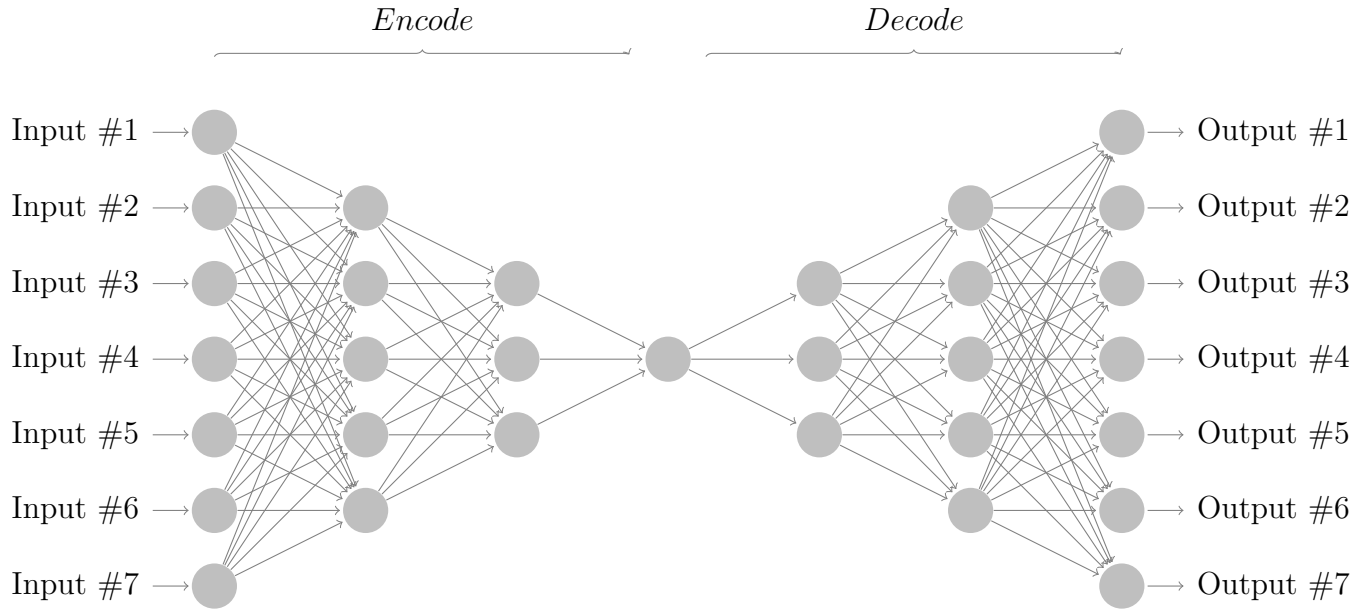


Figure 2.6: Graph depicting a stacked autoencoder.

The dimension of the stacked autoencoder consists of the number of layers and number of neurons in each hidden layer. Later when referring the the dimension of the stacked autoencoder it will be denoted as $[x_1, ..., x_n]$. For figure 2.6 the dimension is thus $[7, 5, 3, 1, 3, 5, 7]$.

To calibrate and train the stacked autoencoder the same techniques is applied as when training any other artificial neural network. The *output error* is measured and is applied to the *loss function*. The loss function determines the distance between the input vector and output vector. Since the aim is to reconstruct the inputs the loss functions should thus be minimised as far as possible because if the loss is zero the input and output vectors are the same. The loss function is then passed onto an optimiser which adjusts the stacked autoencoders weights and biases of its hidden layers neurons appropriate to the *error-gradient*.

2.3.2 Overfitting and underfitting

The dimension of the stacked autoencoder is important because if too large the network might not generalise well given new samples which aren't used for training. This behaviour of not generalising to new samples outside the training sample-space is called overfitting and is a common problem in general for artificial neural networks. Underfitting is the opposite when the autoencoders dimensionality is too small to learn to reconstruct its inputs. [1, p. 26]

2.4 Architectural problems and regularisation

2.4.1 Data feature engineering

Each value in an input vector is called a data feature. The data feature represents a value in the training data and can represent anything. In artificial neural network architectures a huge variation of scale between data features can be hard to model correctly for the network. [1, p. 25] Data features that aren't interesting or that doesn't provide any desirable pattern for

the model can therefor be removed from the sampled data. Transforming the sampled data is referred to as data feature engineering.

2.4.2 Vanishing/exploding gradients

The vanishing gradient problem is when the error-gradient gets consecutive smaller for each layer. This basically leaves the weights and biases in the lower layers of the deep neural network unchanged. Since the weights and biases aren't updated the learning stops, it vanishes. [1, p. 275]

The opposite can also happen, where the gradients can grow larger and larger, which brings the deep neural network farther and farther away from learning the desired patterns. [1]

2.4.3 He-initialisation regularisation

He-initialisation is a regularisation technique meant to reduce the risk of overfitting. [5] This technique helps alleviate the vanishing gradients problem by ensuring that the variance is the same for the input and output of a neuron layer. The same is applied to the gradients during backpropogation. [1, p. 277]

2.4.4 Batch normalisation regularisation

One common regularisation technique is batch normalisation. The technique works by zero-centring and normalising given input data before it is fed through the activation functions of each layer. Doing this returns two output parameters: one for the features value and one for scaling. By doing this batch normalisation can even out the influence different features have upon the deep neural network. [1, p. 282]

3. Method

3.1 Planning

Initially a more general plan of the development and research was created. The plan consisted of less strict milestones with the purpose to give perspective over the projects development. Weekly meetings were held with the projects supervisors and there technical questions were asked and goals of what to achieve until the next meeting.

3.2 Tools

The choice of using TensorFlow was set from the start. TensorFlow is a framework for machine learning implemented with Python made by Google and gives the ability to construct artificial neural networks. TensorFlow supports NVIDIA's CUDA acceleration which gave us the ability to utilise GPUs for the training. TensorFlow supports implementations into multiple programming-languages though the amount of documentation is the most extensive for the Python implementation. [7]

For creating data-sets MatLab was initially used before the data-sets were available from the miniature-scaled research vehicle. [3]

3.3 Retrieving test data

Before development of the application could start, some sort of training data was needed and because the run with the research vehicle was not scheduled until later in the projects time-line some sort of artificial data had to be generated. To generate these artificial data-sets a simulation of a parallel hybrid transmission system was conducted using MatLab. The simulation provided sufficient however simple data-sets which made it possible to start developing the application. [3]

The data-sets used for training and testing in the final results were provided by Benjamin Vedder[8] and his self-driving model vehicle. The vehicles log-data could be used for our network (*see Appendix figure A.1 for image of the vehicle*) and because the vehicle was self-driving fairly consistent runs could be conducted. The runs consisted of multiple lapses around a specified path with multiple different turns wide and quick as well as a variety of speeds with the vehicle accelerating and decelerating.

Four different runs were made two without induced errors and two with. One of the runs without induced error were used for training the network and the other for verification, this run was also conducted on a different path. The first induced error was made by placing a resistance between the vehicles batteries and motor (*see Appendix figure A.2*). The second induced error was produced by lowering the suspension of one of the vehicle's front wheels.

3.4 To train the network

To being able to use a neural network it needs to be properly trained. Training a neural network might take a long time with lots of trial and error before the systems hyperparameters are correctly tuned. Whilst doing this we also learned the importance of having a modular application with an easy way to modify parameters to test different algorithms and values. During this phase of this project different optimisers, losses and other techniques were tested and evaluated.

Another part of training the network was to engineer and adjust features from the given data-set. From the given data set the feature time stamp had to be removed, because it made the network find bad connections which skewed the reconstruction. The GPS data features were also required to be removed because there are no connection between the vehicles geographical position and other features and this would also lead to a skewed reconstruction.

The application TensorBoard[6] were used to monitor the training progress.

3.5 Testing

To find out whether a sample is an anomaly, the ability the autoencoder had reconstructing the input data was analysed. If the delta between the reconstructed and input data differed a lot from the mean delta there was a high probability of an anomaly. The delta can be seen in the result of the sessions loss function. When testing the given samples were run by testing each row serially through the test data-sets.

Two different data-sets were used to test the network. There were also used another no-fault validation data-set in addition to the training data. This was used to validate the results reliability and to ensure that the trained model could generalise error-free data-sets. This ensures that the network is not overtrained, thus not only being able to reconstruct the data that it had been trained on. The tests results were graphed and rendered with TensorBoard[6].

4. Construction

4.1 General structure

The application is built upon two Python-files, `app.py` and `autoencoder.py`. These files takes care of the initialisation, training and testing the final trained model of the autoencoder after training.

The application consists of one class, the autoencoder and the entire project structure is built around it. The project is made up by two Python-files: `app.py` and `autoencoder.py`.

4.1.1 Initialisation

Inside the `app.py` file there is a Python dictionary that acts as a configuration list. This config dictionary handles most hyperparameters of the autoencoder and is specified before training or testing any model. The general structure of the config dictionary and its hyperparameters passed onto the autoencoder can be seen below:

```
config = {
    'file_load': 'data/csvfile.dat',
    'file_save': 'data/csvfile.txt',
    'epoch': hyperparameter,
    'hidden_dim': hyperparameter,
    'num_layers': hyperparameter,
    'batch_size': hyperparameter,
    'starter_learning_rate': hyperparameter,
    'activation': hyperparameter,
    'optimizer': hyperparameter,
    'momentum': hyperparameter
    }
```

4.1.2 Preprocessing and parsing

Before running the autoencoder either in training or testing mode the csv-files specified in the config dictionary are preprocessed. The csv-files are comma separated and the parser in the `app.py` file takes care of preprocessing them into an two dimensional array.

This is done by running the `load_data(...)` and passing it into an array. This preprocessed two dimensional array will later be passed onto the autoencoder depending if the user either wants to train the model or test the model, this can be seen below:

```
data = np.array(load_data(config['file_load']))
```

The code-snippet above preprocesses the specified csv-file. The implementation of `load_data` can be seen in Appendix *figure A.3*.

4.1.3 app.py

The `app.py` file takes care of coordinating all both the initialisation and preprocessing. It contains the code to load the config dictionary to the autoencoder, the `load_data(...)` method and the code which handles the run arguments for the application.

The run-arguments are one of the following three and only one can be specified, to see how to use them and the application see section 4.3. The three run-arguments can be seen below:

```
train new <name of model>
train cont <name of model>
test <name of model>
```

4.1.4 autoencoder.py

The `autoencoder.py` handles the entire construction, training and test operations of the deep neural network. It consists of two main parts, the helper methods and the autoencoder object. There are two different implementations of the autoencoder object inside `autoencoder.py`, one which implements dropout and another that doesn't.

There are several batch generative helper methods. The first of these is a randomizing batch method which implements the idea of stochastic gradient descent training. This method fills the batch with randomised samples from the data fed to the autoencoder when a batch is needed. The implementation can be seen below:

```
def get_batch(X, size):
    a = X[np.random.choice(len(X), size, replace=False)]
    return a
```

The second batch generative helper method randomises a starting index and returns a batch of specified size from that starting index. This method implements the idea of mini-batch training and the implementation can be seen below:

```
def get_randombatch(X, size):
    start = np.random.randint(len(X)-size, high=None, size=None)
    end = start+size-1
    arr = X[start:end:1]
    return arr
```

And lastly is the sequential batch method that returns a batch of specified size from specified starting index. The autoencoder keeps track of the index and fetches batches sequentially, thus the name of the method. This method implements the idea of mini-batch training and the implementation can be seen below:

```
def get_seqbatch(X, size, index):
    end = (size*index)+size-1
    start = index * size
    arr = X[start:end:1]
    return arr
```

There are also two helper methods that specifies which activation functions and optimisers the autoencoder object shall use depending what is specified in the config dictionary, the returned

values from these helper methods are TensorFlows built in implementations. The input to these methods are the hyperparameters from the config dictionary in `app.py`. The purpose of these helper methods if to make the application modular depending on what is specified in the config dictionary.

The first one determines what kind of activation function the autoencoder object shall use and there are only four allowed activation functions, the *ReLU, logistic, ELU and linear*. The implementation can be seen below:

```
def activation_fn(self, activation):
    if activation == 'relu':
        self.scale = False
        return tf.nn.relu
    elif activation == 'sigmoid':
        return tf.nn.sigmoid
    elif activation == 'elu':
        return tf.nn.elu
    elif activation == 'None':
        return None
```

By default the value `self.scale` is set to true for all activation functions besides for the *ReLU* activation function, this is why the `self.scale` is set to false before the *ReLU* is returned.

The second helper method determines what optimisers is used by the autoencoder object depending on what is specified in the config dictionary. The implementation can be seen below:

```
def optimizer_(learning_rate, optimizer, loss, momentum):
    if optimizer == 'MomentumOptimizer':
        return tf.train.MomentumOptimizer(learning_rate=learning_rate,
            momentum=momentum, use_nesterov= True)
    elif optimizer == 'GradientDescentOptimizer':
        return
            tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
    elif optimizer == 'AdamOptimizer':
        return tf.train.AdamOptimizer(learning_rate=learning_rate)
```

To construct the autoencoder there is a helper method that returns an array containing the number of neurons per hidden-layer of the *encoder* which can be indexed during construction. Since the *decoder* has the same dimension as the *encoder* but mirrored the same returned array can be indexed during construction. The implementation can be seen below:

```
def get_dims(self, hidden_dim, num_layers, delta):
    dim = []
    temp = hidden_dim
    for value in range(num_layers):
        if temp >= 0:
            dim.append(temp)
            temp -= delta
        elif temp < 0:
            dim.append(0)
        pass
    print(dim)
    return dim
```

The variable `delta` to the method determines how many neurons each consecutive layer decreases/increases for the *encoder* and *decoder*.

The helper method `printparams(...)` creates a text-file which prints the hyperparameters

of the autoencoder and some other essential parameters such as which csv-files are used for training and testing to a text-file. An example of a parameter text-file can be seen below:

```
10000
Input dimension(also known as number of features): 19
Activation function used: <function relu at 0x0000000005762D08>
Dimensionality of hidden layers: [19, 15, 11, 7, 3]
Number of hidden layers: 5
Number of epochs: 10000
Batch_size: 2000
Starter learning rate: 0.01
Datafile: data/csvNOFAULT_NOGPS.dat
Batch norm params:
        is_training: Tensor("is_training:0", shape=(), dtype=bool)
        decay: 0.99
        updates_collections: None
        scale: False
Optimizer: 'AdamOptimizer'
```

The implementation for the `printparams(...)` method can be seen in Appendix figure A.4. The first integer value in the parameter text-file is the epoch offset, which simply stats at which epoch the model should start.

Lastly the method `getOffset(...)` returns the epoch offset of a saved models parameter text-file. This method is used when the autoencoder loads an old model to continue training.

In the next chapter the construction of the two different autoencoder will be presented by showing how the class "autoencoder" inside the `autoencoder.py` file is coded.

4.2 Constructing the autoencoder using TensorFlow

To perform a simulation with TensorFlow a session object is required. The session object keeps track of the current model and can be saved or loaded. This means that when a TensorFlow function is executed through a session, with the `session.run` function, it will utilise the properties of the currently loaded model. These properties are for example weights or biases values. Because of the of the session object, TensorFlow applications have to initialise all functions and operations before running them.

High-level TensorFlow API functions were used to create the network, and the function `fully_connected()` retrieved from the `contrib` framework within TensorFlow simplified the implementation. This method creates a fully connected layer of neurons as well as applies all needed regularisation like weight initialisation and batch normalisation, the function also returns a shape with the output from the layers neurons. The layer being fully connected, each neuron gathers input from a specified shape, in the examples instance `x`, which gives the ability to Daisy-chain the layers together. This makes it possible for us to build a network. An example of the method is shown below, this version doesn't implement dropout:

```
encoded1 = fully_connected(x, self.layers[0],
    weights_initializer=he_init, scope="eh1", normalizer_fn=batch_norm,
    normalizer_params=self.bn_params, activation_fn=self.activation)
```

The autoencoder object uses the method `get_dims(...)` to get the desired layer sizes. This is used to create the *encoder* and *decoder* by indexing `self.layers[...]`, the autoencoder has a similar shape as *figure 2.6*. As mentioned earlier the `fully_connected` function is used to create

the neurons layers and all regularisation methods are passed into this method. Below is the an implementation of an `fully_connected` layer but with dropout regularisation as well:

```
encoded1 = fully_connected(X_drop, self.layers[0],
    weights_initializer=he_init, scope="eh1", normalizer_fn=batch_norm,
    normalizer_params=self.bn_params, activation_fn=self.activation)
encoded1_dropout = tf.contrib.layers.dropout(encoded1, keep_prob,
    is_training=self.is_training)
```

During construction the optimiser used by the autoencoder is decided by calling the helper function `optimizer_(...)` which returns the TensorFlow optimiser specified by the config dictionary. For the optimiser to work a loss function needs to be defined. Below is the code-snippet for the initialisation of optimiser and loss function of the autoencoder version without dropout:

```
loss = tf.losses.mean_squared_error(predictions=logits, labels=self.x)
optimizer = optimizer_(self.starter_learning_rate, self.optimizer,
    loss, self.momentum)
training_op = optimizer.minimize(loss)
```

And the equivalent code-snipped for the version with dropout can be seen below:

```
loss = tf.losses.absolute_difference(predictions=logits, labels=self.x)
optimizer = optimizer_(self.starter_learning_rate, self.optimizer,
    loss, self.momentum)
training_op = optimizer.minimize(loss)
```

4.2.1 Train

The autoencoder holds two public functions which makes it possible to interact with the network, the first one is `train(...)`. The `train(...)` function trains the network by running a session, either loaded or new. The network will then train for a specified amount of epochs, logging data to TensorBoard simultaneously at certain intervals. To get a batch during training the desired batch function is called. Below is the code-snippet highlighting the vital parts of training used for both versions of the autoencoder:

```
for i in range(self.epoch):
    ...
    for batch_index in range(n_batches):
        batch_data = get_seqbatch(np.array(data), self.batch_size,
            batch_index)
        l = sess.run([self.loss], feed_dict={self.is_training: False,
            self.x: batch_data})
        _ = sess.run([self.training_op], feed_dict={self.is_training:
            True, self.x: batch_data})
        if batch_index % n_batches == 0:
        ...
        if batch_index % 10 == 0:
        ...
    pass
```

For the entire implementation of the `train(...)` function see Appendix *figure A.4*.

To train the network the session can simply run the training operation, `training_op` which has been previously initialised, the variable `feed_dict` specifies which batch should be inserted as input to the network. This can be seen below:

```
_ = sess.run([self.training_op], feed_dict={self.is_training: True,
    self.x: batch_data})
```

In TensorBoard the progress will be presented as a graph with average loss as y-axis and batch index as x-axis. The session will save the model when the training is done or stopped.

4.2.2 Test

The function `test` evaluates any loaded data-set against the trained autoencoder model currently loaded into the TensorFlow session. It will iterate over the loaded data-set, running the loss function and logging the result to TensorBoard. In TensorBoard the result will be represented as a graph with loss as y-axis and x-axis as the number of samples of the test data-set. Below is the code-snippet highlighting the vital parts of the `test(...)` function:

```
def test(self, data):
        with tf.Session() as sess:
            ...
            for i in range(len(hidden)):
                loss = sess.run([self.loss],
                    feed_dict={self.is_training: False,self.x:
                    [data[i]]})
                summary_str =
                    self.test_summary.eval(feed_dict={self.is_training:
                    False, self.x: [data[i]]})
                self.file_writer.add_summary(summary_str, i)
                pass
            print("Done with test.")
```

For the entire implementation of the `test(...)` function see Appendix *figure A.5*.

4.3 How the application is used

Training

To train run:

```
python app.py train new <name of model>
```

To continue training the current model run:

```
python app.py train cont <name of model>
```

Testing

To test run:

```
python app.py test <name of model>
```

To utilise TensorBoard:

```
tensorboard --logdir tf_logs/ hosted at localhost:6006
```

5. Discussion

5.1 Network dimensionality

One important aspect of creating an artificial neural network is to choose a proper amount of layers and neurons. By compressing the data down to a few or even a single neuron in the central layer of the autoencoder lowers the risk of the networks not generalising the training data well enough. As Aurelien Geron claims in the book Hands-On Machine Learning with Scikit-Learn & TensorFlow [1, p. 275] we also found that the deeper the network gets the more time it took for the network to converge to a good solution. This forces a compromise having accuracy or low convergence time. In our case because of the low amount of features fed into the network the amount of layers did not matter in any significant way and is the reason why we settled for a rather arbitrary dimension of ten layers for most of our tests.

5.2 Batch size

The amount of samples run through the network during training affected the networks accuracy and produced a relatively volatile learning curve. By increasing this during training a more stable learning curve was achieved, this is clearly visible in figure 5.1. Even though having a higher batch size provided a more stable model it also requires a longer training time before converging. When increasing batch sizes diminishing returns also appears, this is seen in *figure 5.1* where the difference between a batch size of 1000 and 2000 is more distinctive than the difference between 2000 and 5000.

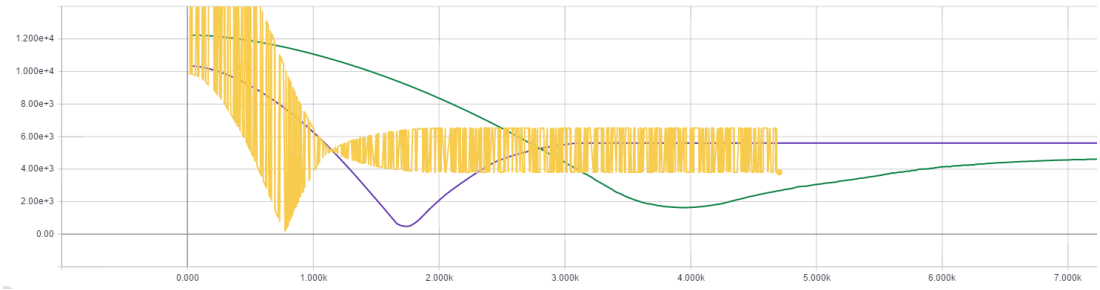


Figure 5.1: Yellow: 1k, Blue: 2k, Green 5k

5.3 Training time

Meanwhile training a network it is important to monitor the learning curve. When performing unsupervised anomaly detection with an autoencoder the network gains the ability to reconstruct values first the the learning curve has converged around a constant value.

5.4 Optimizers

Because Adam optimisation was easy to implement and provides a more adaptive learning[1, p. 299], it was mainly used for the models being generated through this project. Other optimisers such as *Gradient Decent* or *Momentum* might provide a sufficient result training the network, though this was not explored in this project just because of the ease of implementation and reliability of Adam.

5.5 Activation function

Choosing which type of activation-function can be a tricky matter. Luckily the process of implementing different activation-functions is fairly simple in TensorFlow which made it possible for us to have the time to test different functions. The functions *ELU* and *ReLU* performed way better than the other functions when it comes to ability to learn and how long time it takes. In the *figure 5.2* below it is possible to see the learning curve of the same model with different activation-functions and though the messy figure it is possible to see that neither the sigmoid, tanh or softmax activation functions reaches an acceptable loss in reasonable time. This is the main reason why *ReLUs* and *ELUs* where used in this projects final simulations.

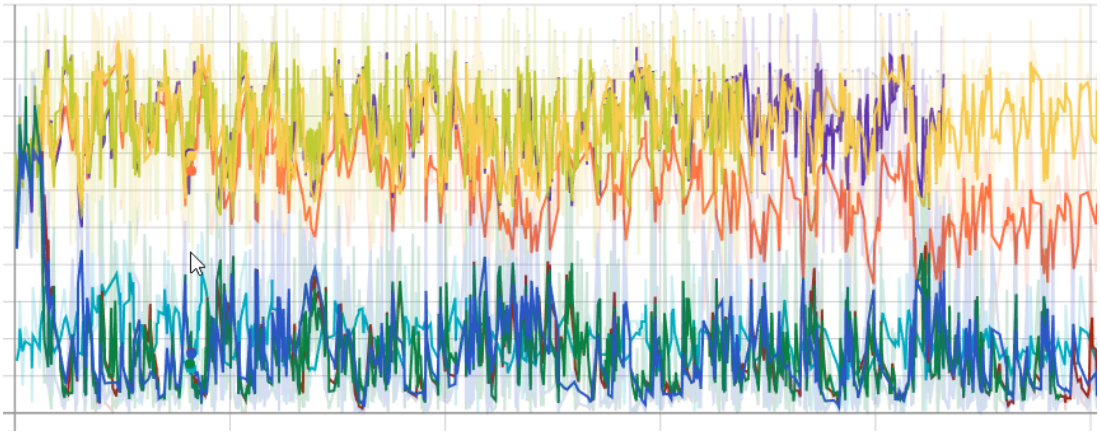


Figure 5.2: Lower is better, Blue: ReLU, Yellow: logistic, Light green: tanh, Purple: softmax, Red: ELU, Orange: relu6

5.6 Batch Normalisation

Because the given features varying in span and scale the use of batch normalisation helped the network and made it converge at a lower loss than previously. At the beginning of the project the use of the logistic function was favoured and as visible in *figure 5.2* it gave quite a larger loss than other activation function. This was the cause because the features in the data-sets were of varying scale, for the logistic function to work properly the features needs to be scales between 0 and 1. Batch normalisation takes care of this issue.

5.7 Weight and bias initialisation techniques

The effect of using the regularisation techniques Xavier- or He-initialisation had on the networks is that it constrains and simplifies the network by reducing the degrees of freedom the network has. It lowered the initial loss during training quickening the convergence.

There was no definitive difference between Xavier- and He-initialisation for our network, perhaps when having different feature sets it would be something to consider. He-initialisation appeared to perform marginally quicker though this could just be resulted by the fact of different runs perform at different speeds in general.

6. Result

The results are comparative to different models which explores different hyperparameters of the deep autoencoder. The results of the different models all have a similar form in relation to the training and test data-sets.

6.1 Trained Model

The first autoencoder model that gave valid results, referred to as M-1 utilises the following hyperparameters:

$$
\begin{aligned}
&Optimiser && -\text{ Adam}\\
&Activationfunction && -\text{ ELU}\\
&Lossfunction && -\text{ MSE}\\
&Batch-size && -\text{ 2000}\\
&Learnrate && -\text{ 0.001}\\
&Dimension && -\text{ [19,15,11,7,3]}\\
&Dropout && -\text{ No}
\end{aligned}
$$

The Dimension hyperparameter refers to the number of hidden-layers and neurons per hidden layer, nineteen neurons for the first hidden layer, fifteen neurons for the second and so on, on both the decode and encode sides of the autoencoder.

After construction of the M-1 the training-phase was initialised and the loss was plotted over time which can be seen in *figure 6.1*. The loss converged towards a stable value after 30000-40000 epochs at a value of 600 million, see *figure 6.1*. After the training phase the autoencoder is fed the testing data-sets to evaluate if it actually is able to generalise new data, this can be seen in *figure 6.3*.

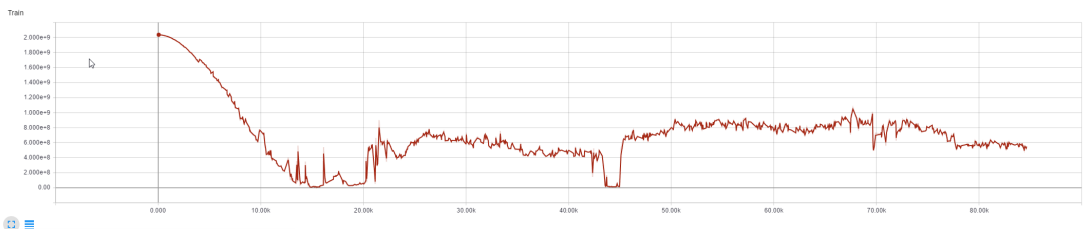


Figure 6.1: M-1 during training-phase

All autoencoders following the experiments of M-1 utilised dropout to ensure that the autoencoders weren't overfitting the training data-set. The shared hyperparameters of the other autoencoders, M-2, M-3 are:

$$
\begin{aligned}
&Optimiser && - \text{Adam} \\
&Activationfunction && - \text{ReLU} \\
&Lossfunction && - \text{MAE} \\
&Learnrate && - 0.01 \\
&Dimension && - [19,15,11,7,3] \\
&Dropout && - \text{Yes}
\end{aligned}
$$

M-2 and M-3 explores the difference batch-sizes has upon the loss during the training-phase of the autoencoders, see *figure 6.2* as well as how well the M-2 and M-3 model generalises during the testing-phase of testing data-sets, see *figure 6.5* and *figure 6.6*. The batch-sizes for the two autoencoders are:

$$
\begin{aligned}
M-2 : Batch-size-2000 \\
M-3 : Batch-size-5000
\end{aligned}
$$

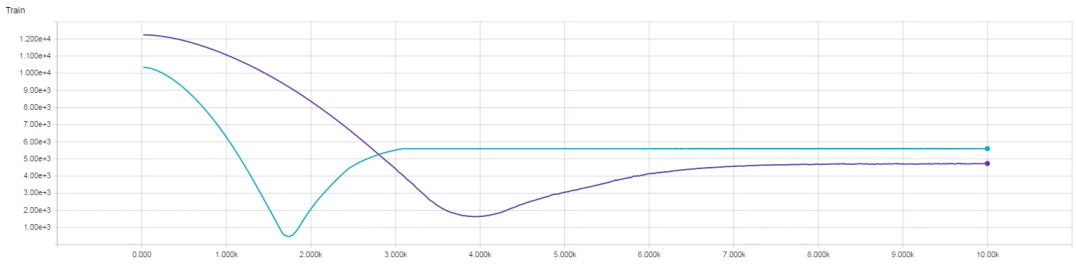


Figure 6.2: Turquoise: M-2, Purple: M-3 during training-phase

The M-2 converges towards a loss value of 5600 and the M-3 converges towards a loss value of 4700.

6.2 Test results

After the training-phase the test data-sets are used to firstly validate that the model can generalise to new samples. When plotting the following different testing data-sets:

- *Training data-set*
- *Validation data-set*
- *Erroneous resistance data-set*
- *Erroneous suspension data-set*

There is a clear visible difference between them. In *figure 6.3* is the plotted testing data-sets for the M-1 model, the yellow graph represents the training data-set, even known as the generalised model. The purple graph is the validation data-set and has the lowest loss, meaning it reconstructs its inputs closest to the generalised model. The red and orange graph however, respectively the erroneous resistance and suspension data-sets, have a quite larger reconstructed loss. As visible between these graphs the model can more easily reconstruct the data-sets without any induced errors in relation to the generalised model compared to the ones with induced errors.

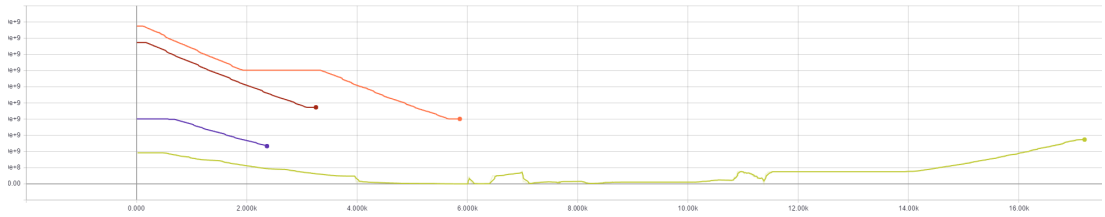


Figure 6.3: Green: Training, Purple: Validation, Red: Err. resistance, Orange: Err. suspension for M-1.

The same tests were done for M-2 and M-3 illustrating their ability to reconstruct the validation data-set and the erroneous data-sets. As visible in *figure 6.4 and 6.5* for both M-2 and M-3 they have very similar shape and the effect of the batch size is minimal for the final generalised model. They are both able to detect the induced anomaly in the validation data-set. There is also quite a considerable similarity between the generalised model of M-1 and those of M-2 and M-3.

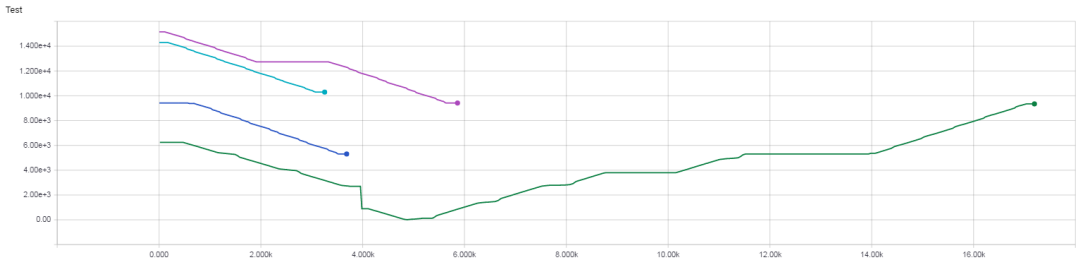


Figure 6.4: M-2. Green: Training, Blue: Validation, Light-blue: Err. resistance, Purple: Err. suspension.

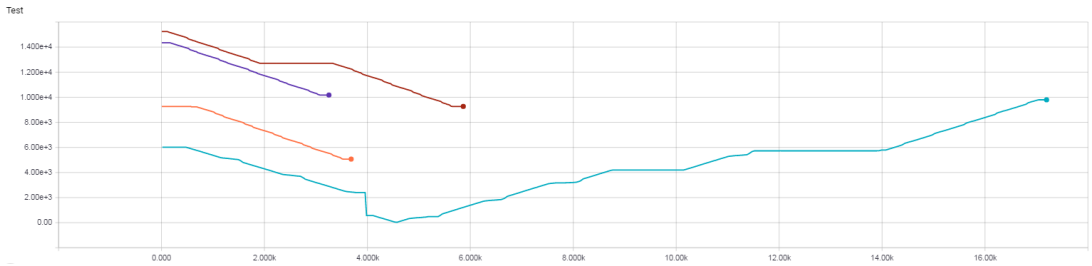


Figure 6.5: M-3. Turquoise: Training, Orange: Validation, Blue: Err. resistance, Red: Err. suspension.

7. Conclusion

It is possible to use unsupervised anomaly detection to find erroneous or anomalistic behaviour in data-sets similar to data-sets retrieved from real electrical architectures of automotive vehicles. An application has been constructed which provided with the generated test-data could prove this concept, finding a difference between data-sets with no errors and data-sets with induced errors. Though it is important to note that this project should only be seen as a proof of concept and not a real testament of the effects unsupervised anomaly detection can have in the context of development and error detection.

7.1 Applications

This method could be applicable in not only specifically electrical vehicle architectures but also in a more general sense. If a situation where a lot of data is being logged unsupervised anomaly detection could be applied and used for finding errors or unexpected behaviour. This could hypothetically save time for developers finding bugs or suspicious behaviour within their systems.

7.2 Further explorations

Taking from whats been learned from this project further research could be done. Some efforts gathering more extensive data-sets with multiple different induced errors and different paths. This would be done to validate the reliability of the network. By training the network on more diverse and valid data-sets, the model might be able to generalise to increase the accuracy of detecting anomalies which can occur in a wide range of diverse scenarios.

Another area of exploration of this project is to run different combinations of hyperparameters and components of the autoencoder. Exploring new and different data-sets might require different models with different hyperparameters and components to create well generalising model. A bad autoencoder model might also take too long to converge to a good generalisation so a ”quick-learner” would be preferable.

It would be interesting to test the method on other types of system and data. For example server equipment or other electrical systems with similar properties.

Bibliography

[1] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reily, 2017.

[2] David Kriesel. *A Brief Introduction to Neural Networks*. David Kriesel, 2007.

[3] MathWorks. June 2017. URL: https://se.mathworks.com/help/physmod/sdl/examples/parallel-hybrid-transmission.html?requestedDomain=se.mathworks.com.

[4] Andres Munoz. June 2017. URL: https://www.cims.nyu.edu/~munoz/files/ml_optimization.pdf.

[5] Michael Nielssen. *Neural networks and deep learning*. June 2017. URL: http://neuralnetworksanddee com.

[6] Tensorboard. June 2017. URL: https://www.tensorflow.org/get_started/summaries_and_tensorboard.

[7] Tensorflow. June 2017. URL: https://www.tensorflow.org/.

[8] Benjamin Vedder. *The RISE Self-Driving Model Vehicle Platform (SDVP)*. June 2017. URL: https://github.com/vedderb/rise_sdvp.

[9] Vinnova. *FFI - Fordonsstrategisk Forskning och Innovation*. June 2017. URL: http://www.vinnova.se/sv/ffi/.

Appendix

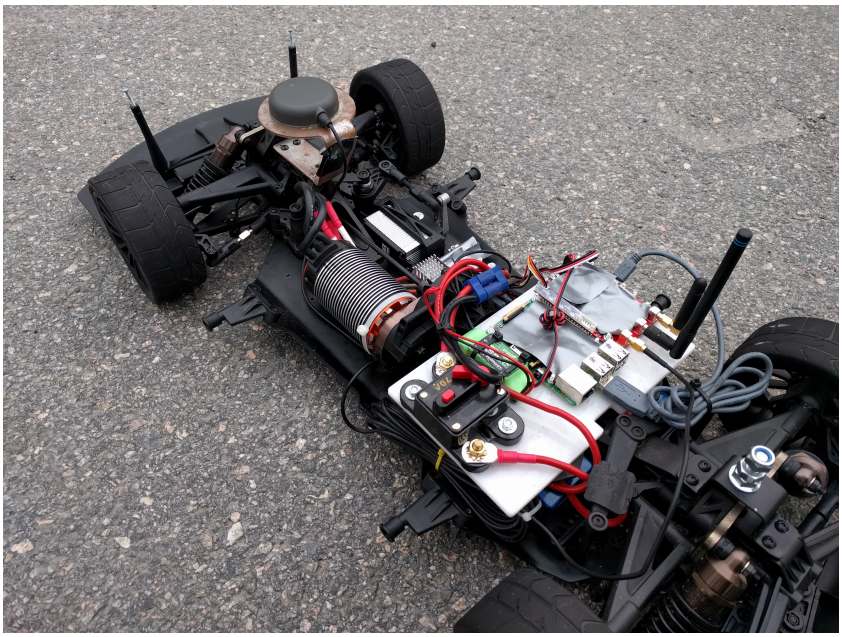


Figure A.1: car without resistance

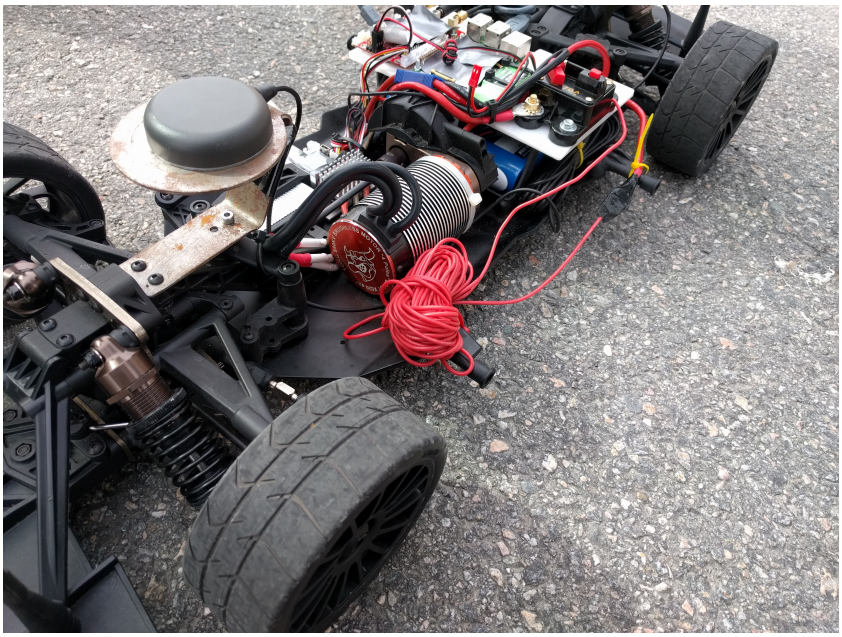


Figure A.2: car with resistance

```
def load_data(filename_load):
    #print status
    print("Fetching data from %s"%(filename_load))

    #opens file and defines variables
    loaded_file = open(filename_load, "r")
    data = []

    #reads from file
    for line in loaded_file:
        #removes whitespace and splits code
        read = line.strip("\n").split(",")
        tmpline = []

        #format data to correct data type
        for x in read:
            tmpline.append(float(x))
            pass

        #appends data to final array
        data.append(tmpline)
        pass

    #close file after it has been read
    loaded_file.close()

    return data
```

Figure A.3: Implementation of method `load_data`

```
def train(self, data, cont=True, dir=None):
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        n_batches = int(np.ceil(len(data) / self.batch_size))

        currentEpochOffset = 0
        currentStep = 0

        if cont:
            self.saver.restore(sess,
               './{}/final_model.ckpt'.format(self.logdir))

            currentEpochOffset = getOffset(self.logdir)
        try:
            currentStep = currentEpochOffset
            for i in range(self.epoch):
                currentStep += 1
                for batch_index in range(n_batches):
                    batch_data = get_seqbatch(np.array(data),
                       self.batch_size, batch_index)
                    l = sess.run([self.loss],
                       feed_dict={self.is_training: False, self.x:
                       batch_data})
                    _ = sess.run([self.training_op],
                       feed_dict={self.is_training: True, self.x:
                       batch_data})
                    if batch_index % n_batches == 0:
                        data_lr = self.starter_learning_rate
                        string = 'epoch {0} : \t loss = {1} \t learning
                           rate = {2}'.format(i+currentEpochOffset, l,
                           data_lr)
                        print(string)
                    if batch_index % 10 == 0:
                        summary_str =
                           self.mse_summary.eval(feed_dict={self.is_training:
                           False, self.x: batch_data})
                        self.file_writer.add_summary(summary_str, i +
                           currentEpochOffset)
                pass

        except KeyboardInterrupt:
            print('Aborted learning... saving model')
            self.saver.save(sess,
               './{0}/final_model.ckpt'.format(self.logdir))
            self.printparams(currentEpochOffset=currentStep)
            try:
                sys.exit(0)

            except SystemExit:
                os._exit(0)

        self.saver.save(sess,
           './{0}/final_model.ckpt'.format(self.logdir))
        self.printparams(currentEpochOffset=currentStep)
```

Figure A.4: Implementation of method `train`

```
def test(self, data):
    with tf.Session() as sess:
        self.saver.restore(sess,
            './{0}/final_model.ckpt'.format(self.logdir))
        lenght = len(data)
        print("Data lenght: %r" % lenght)
        hidden,reconstructed = sess.run([self.encoded, self.decoded],
            feed_dict={self.is_training: False, self.x: data})
        loss = sess.run([self.loss], feed_dict={self.is_training:
            False, self.x: [data[1]]})
        for i in range(len(hidden)):
            loss = sess.run([self.loss], feed_dict={self.is_training:
                False,self.x: [data[i]]})
            print(i, '. loss: ', loss)
```

Figure A.5: Implementation of method `test`