Degree project in applied mathematics
Second level, 30 HP

Neural Ordinary Differential Equations for Anomaly Detection

JON HLÖDVER FRIDRIKSSON AND ERIK ÅGREN

Stockholm, Sverige 2022
Neural Ordinary Differential Equations for Anomaly Detection

JÓN HLÖDVER FRIDRIKSSON
ERIK ÅGREN

Master’s Programme, Applied and Computational Mathematics, 120 credits
Date: January 4, 2022

Supervisor KTH: Pierre Nyquist
Supervisor Scania: Juan Carlos Andresen Eguiluz
Examiner: Pierre Nyquist
   School of Engineering Sciences
Host company: Scania CV AB
Abstract

Today, a large amount of time series data is being produced from a variety of different devices such as smart speakers, cell phones and vehicles. This data can be used to make inferences and predictions. Neural network based methods are among one of the most popular ways to model time series data. The field of neural networks is constantly expanding and new methods and model variants are frequently introduced. In 2018, a new family of neural networks was introduced. Namely, Neural Ordinary Differential Equations (Neural ODEs). Neural ODEs have shown great potential in modelling the dynamics of temporal data. Here we present an investigation into using Neural Ordinary Differential Equations for anomaly detection. We tested two model variants, LSTM-ODE and latent-ODE. The former model utilises a neural ODE to model the continuous-time hidden state in between observations of an LSTM model, the latter is a variational autoencoder that uses the LSTM-ODE as encoding and a Neural ODE as decoding. Both models are suited for modelling sparsely and irregularly sampled time series data. Here, we test their ability to detect anomalies on various sparsity and irregularity of the data. The models are compared to a Gaussian mixture model, a vanilla LSTM model and an LSTM variational autoencoder. Experimental results using the Human Activity Recognition dataset showed that the Neural ODE based models obtained a better ability to detect anomalies compared to their LSTM based counterparts. However, the computational training cost of the Neural ODE models were considerably higher than for the models that only utilise the LSTM architecture. The Neural ODE based methods were also more memory consuming than their LSTM counterparts.

Keywords

Anomaly detection, Neural ordinary differential equations, Statistical modelling, Autoregression, Variational autoencoder, Multivariate time series
ii | Abstract
Sammanfattning


Nyckelord

Anomalidetektion, Neurala ordinära differentialekvationer, Statistisk modellering, Autoregression, Variational autoencoder, Multivariat tidsserie
iv | Sammanfattning
Acknowledgements

We are very grateful to have been given the opportunity to write our master thesis at Scania CV AB. The time during the thesis project has been an educational experience with lots of new knowledge within different fields of applied mathematics and insight into how the company operates. We would like to give our gratitude to our supervisor at Scania, Juan Carlos Andresen Eguiluz, for continuous support and feedback throughout the project. We would also like to thank our supervisor at KTH, Pierre Nyquist, for providing us with fruitful discussions and help when needed. Lastly, we want to show our appreciation to the Edge team at Scania for supplying us with fika (coffee break/snacks) every week, giving us guidance as well as insight into their work at Scania.

Stockholm, January 2022
Jón Hlöðver Friðriksson and Erik Ågren
vi | Acknowledgements
Contents

1 Introduction 1
  1.1 Background .......................... 1
  1.2 Problem description ...................... 3
  1.3 Methodology .......................... 3
  1.4 Delimitations .......................... 4
  1.5 Outline .............................. 4

2 Theory 6
  2.1 What is an anomaly? ...................... 6
  2.2 Mathematical Background ................. 8
    2.2.1 Gaussian Mixture Model .............. 9
    2.2.2 Feedforward Neural Networks ......... 11
    2.2.3 Recurrent Neural Networks .......... 15
    2.2.4 Residual Networks .................. 19
    2.2.5 Neural Ordinary Differential Equations 21

3 Methods 29
  3.1 Methods to detect anomalies ............... 29
    3.1.1 Anomaly score .................... 30
  3.2 Benchmarking .......................... 32
    3.2.1 Receiver Operating Characteristics 32
  3.3 Data ................................ 34
    3.3.1 Human Activity Recognition ......... 34
  3.4 Model Choice ......................... 37
  3.5 Experimental Settings ................... 37
    3.5.1 Probabilistic method ............... 38
    3.5.2 Autoregressive methods ............. 38
    3.5.3 Variational autoencoders .......... 38
    3.5.4 Settings for the neural ODE based methods 39
4 Results 40
4.1 Probabilistic Method .......................... 40
4.2 Autoregressive Methods ..................... 41
4.3 Variational Autoencoders ................. 43
4.4 Summary ..................................... 44

5 Discussion 47

6 Conclusions 50
6.1 Future Work ................................. 51

References 53

A SKAB data 57
A.1 The data .................................. 57
A.2 Results .................................. 59
A.2.1 Probabilistic Method .................. 59
A.2.2 Autoregressive methods ............. 60
A.2.3 Variational autoencoders .......... 61
A.3 Discussion ............................... 63
# List of Figures

2.1 Three types of anomalies ................................................. 7  
2.2 Roadmap showing relations between the different methods .... 9  
2.3 A single Layer Perceptron ........................................... 12  
2.4 A multi-Layer Perceptron ........................................... 14  
2.5 Inputs and outputs from an RNN model ......................... 16  
2.6 The architecture of an RNN cell ................................ 17  
2.7 The architecture of an LSTM cell ................................ 19  
2.8 A residual block ..................................................... 20  
2.9 Trajectories of ResNet and Neural ODE ......................... 23  
2.10 A comparison between a standard Recurrent neural network (RNN) and Ordinary Differential Equation (ODE)-RNN .... 25  
2.11 The dynamics of a latent-ODE model .......................... 27  
2.12 The architecture of a variational autoencoder ............... 28  
3.1 Example of ROC-curves for detectors of varying performance 33  
3.2 A series of accelerometer readings from the Human Activity Recognition dataset ..................................................... 36  
4.1 ROC-curves generated by a Gaussian mixture model ......... 41  
4.2 ROC-curves generated by an LSTM model ...................... 42  
4.3 ROC-curves generated by an LSTM-ODE model ............... 42  
4.4 ROC-curves generated by an LSTM variational autoencoder model .............................................................. 43  
4.5 ROC-curves generated by a Latent-ODE model ............... 44  
4.6 Summary of the AUC-scores ........................................ 45  
A.1 Anomaly in the SKAB data ......................................... 58  
A.2 ROC-curves generated by a Gaussian mixture model on the SKAB data .......................................................... 60  
A.3 ROC-curves generated by an LSTM on the SKAB data ....... 60
A.4 ROC-curves generated by a LSTM-ODE on the SKAB data. . 61
A.5 ROC-curves generated by a LSTM-VAE on the SKAB data. . 62
A.6 ROC-curves generated by a Latent-ODE on the SKAB data. . 62
List of Tables

2.1 Common activation functions .......................... 13
2.2 Relationship between some ANNs and integration scheme for Neural ODEs .................................. 22
3.1 Features of the Human Activity Recognition dataset. .......... 35
3.2 Activities of the Human Activity Recognition dataset. ......... 35
4.1 Comparison of the different models for HAR data based on the number of parameters and training time relative to the LSTM model .................................................. 46
A.1 Features of the SKAB dataset. ............................ 58
List of acronyms and abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ANN</strong></td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td><strong>DNN</strong></td>
<td>Deep Neural Network</td>
</tr>
<tr>
<td><strong>ELBO</strong></td>
<td>evidence lower bound</td>
</tr>
<tr>
<td><strong>GMM</strong></td>
<td>Gaussian Mixture Model</td>
</tr>
<tr>
<td><strong>GRU</strong></td>
<td>Gated recurrent unit</td>
</tr>
<tr>
<td><strong>IoT</strong></td>
<td>Internet of Things</td>
</tr>
<tr>
<td><strong>LSTM</strong></td>
<td>Long Short-Term Memory model</td>
</tr>
<tr>
<td><strong>Neural ODE</strong></td>
<td>Neural Ordinary Differential Equation</td>
</tr>
<tr>
<td><strong>NFE</strong></td>
<td>Number of Function Evaluations</td>
</tr>
<tr>
<td><strong>ODE</strong></td>
<td>Ordinary Differential Equation</td>
</tr>
<tr>
<td><strong>PDF</strong></td>
<td>Probability Density Function</td>
</tr>
<tr>
<td><strong>ResNet</strong></td>
<td>Residual Network</td>
</tr>
<tr>
<td><strong>RNN</strong></td>
<td>Recurrent neural network</td>
</tr>
<tr>
<td><strong>VAE</strong></td>
<td>Variational autoencoder</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Increased access to computing power and connectivity comes with possibilities and pitfalls. The Internet of Things (IoT) revolution has seen connected devices introduced into everything from household items and clothes to vehicles and infrastructures (Mattern & Floerkemeier 2010). These devices can collect vast amounts of data. By streaming this data and uploading it to a centralised server, insight like trends and patterns in the data can be utilised to improve the products, monitor their status and safety. However, the amount of data collected, data privacy laws, and potentially limited internet connection pose problems. It is hard to upload all of the data fast enough to not pile up and in a secure manner so that it complies with privacy laws, all while having a limited internet connection.

A way to circumvent uploading the data to a central server is edge computing. Edge computing is the mindset of keeping the data storage and computations as close to the data source as possible. Moving the computation closer to the data source can reduce bandwidth, latency and overhead for the centralised server (Cao et al. 2020). The locally sampled data can be aggregated by each device and used to train many decentralised models. The model parameters from these devices can then be uploaded to a centralised server and combined to update a global model incrementally. This approach of combining local models from many devices to create one global model is called federated learning (McMahan et al. 2017). The model parameters take up less memory than the sampled data. Therefore, uploading only the parameters saves bandwidth. However, since the computing power of the devices is usually limited, constraints are imposed on the utilised model. A
larger model can learn more complex dynamics but at the cost of having more internal parameters, thus being more expensive to train. A smaller model is easy to train and takes up less memory but will perform worse.

A similar trade-off can be found with the data. Sampling at a higher frequency provides more data and can possibly better capture the underlying dynamics. On the other hand, a lower sampling frequency means less data to be stored and faster training. In both cases, failed samples can occur, that further decrease the amount of data and makes the resulting time-series irregular. Therefore, a small model which can perform well on less data and is robust to irregularities in the sampling is preferable.

When collecting data from a data source, anomalies might occur. Those are data points that deviate from their predicted pattern. These anomalous data points may potentially stem from an abnormal behaviour of the source, which may indicate that some error has emerged within the underlying system. For example, an anomalous result of a urine sample can be due to the patient having kidney disease, unusual activity in the data traffic to a server can be caused by a cyberattack, and abnormal sensor readings from a vehicle can stem from a damaged component within the system (Chandola et al. 2009). By detecting the anomalous data and taking the proper action early, the underlying system can be prevented from fatally breaking or being severely damaged.

The benefits of an efficient anomaly detection method from a social, economic and sustainability standpoint are substantial. However, there is not a single method that is state of the art within all different fields. In reality, anomaly detection methods are usually tailored for specific use cases since distinct requirements are imposed on the model depending on the problem at hand. Requirements can be interpretability, computation speed, memory efficiency, robustness to irregularities in the data sampling and low error rate in classifying anomalies. However, there is usually a trade-off between these requirements since a more complex model is often better at detecting anomalies but will be less interpretable and slower. Various methods have been tried to perform anomaly detection, ranging from purely statistical approaches to machine learning methods.

The field of machine learning and neural networks is rapidly growing. As new methods are introduced, it is important to experiment and see if they can improve upon the older methods in various tasks such as anomaly detection. In a recent paper from 2018, Chen et al. (2019) introduced a new family of neural network models called Neural Ordinary Differential Equations (Neural ODEs). These models differ from a plain neural network model in that instead of explicitly specifying the number of hidden layers, the derivative of the
hidden intermediate state is parameterised using a neural network. Neural ODEs are memory-efficient and can naturally model irregularly sampled time series data (Rubanova et al. 2019). Furthermore, from the fact that some ODE solvers can adapt their integration strategy to obtain a result within a given error tolerance bound, the Neural ODE model can have a trade-off between accuracy and computation speed. Therefore, Neural ODE could be a suitable model for anomaly detection within the framework of edge computing.

1.2 Problem description

In this project, Neural ODE models will be implemented for anomaly detection on multivariate time-series data. Their performance will then be compared to the performance of two other well-known methods. The research question is:

Can a neural ordinary differential equation model better perform anomaly detection for multivariate time-series data compared to a Long Short-Term Memory model (LSTM) and a Gaussian Mixture Model (GMM)?

1.3 Methodology

To answer the research question, a suitable dataset will be introduced. Clear tasks will be defined based on this dataset which can accurately compare how the methods (Neural ODE, LSTM and GMM) perform at anomaly detection. Each of the methods will be implemented and individually tuned to make them perform as well as possible for each given task. This tuning will consist of testing different model compositions, hyperparameter tuning and evaluation of different anomaly detection methods for each model type. The different model compositions for these three methods will be obtained by conducting a literature study.

To compare the different models we need some metrics that give a quantitative measure of their performance. To measure the ability to detect anomalies, receiver operating characteristic (ROC) curves along with histograms will be used. ROC curves visually represent the models’ ability to correctly classify points as normal or anomalous based on a discrimination threshold value. To compare the memory cost of each model, the number of internal parameters will be assessed. To evaluate the time it takes to train each model, the training time will be recorded and compared.
The models will be utilised for different experimental settings using multivariate time series data where the sparsity and irregularity of the data will vary.

1.4 Delimitations

For this project, the main focus will be on Neural ODEs and their feasibility for anomaly detection for multivariate time series.

Since Neural ODEs were first introduced in Chen et al. (2019), much work has been done to improve and build upon the method described in the paper. While it would be interesting to see how these extensions of Neural ODEs would perform at detecting anomalies, this project will only look at how the vanilla Neural ODE based models perform. The results will then serve as a first indication of the performance of Neural ODEs at anomaly detection.

As mentioned above, anomaly detection is task-specific. This thesis will only look at anomaly detection for multivariate time-series of sensor data. The experiments conducted will indicate how well Neural ODEs are suited to this specific type of anomaly detection. The results are not necessarily indicative of it’s performance for anomaly detection in other types of data.

Given the limited time for this project, only basic data-preprocessing was done on the datasets, and little hyperparameter tuning was possible. Given more time, we would have liked to do more thorough data exploration and a more thorough search for optimal hyperparameters.

1.5 Outline

- Chapter 2 gives a theoretical background that provides a foundation to understand the different concepts included in the project. This includes a description of the concept of anomalies. The chapter also introduces the mathematical theory of the implemented models. That includes Artificial Neural Networks, Residual Networks, Neural Ordinary Differential Equations, Recurrent Neural networks, Long Short-Term Memory model and Gaussian Mixture model.

- Chapter 3 gives the reader an insight into the methods used to answer the research question. This includes the metrics for comparisons, architecture of the machine learning models, any assumptions made, the data used and the data preprocessing.
• Chapter 4 illustrates the results obtained when conducting the experiments described in the previous chapter. These results act as a foundation to answer the research question.

• Chapter 5 analyses and discusses the results and their implications. The conclusions that can be inferred from the results are addressed in chapter 6.
Chapter 2

Theory

This chapter provides basic background information about anomalies and anomaly detection. Additionally, this chapter describes the models used for anomaly detection in this report, and their mathematical foundation.

2.1 What is an anomaly?

There is no clear definition of what an anomaly is. However, a common description is:

Anomalies are patterns in data that do not conform to a well-defined notion of normal behaviour (Chandola et al. 2009).

This description can seem unclear since the notion of normal behaviour is vague. This raises a question: is it possible to train a machine learning model to find these patterns, and by doing so, learn the normal behaviour of the data?

There are some common problems that make this task challenging:

1. It is difficult to define a threshold boundary at which normal and abnormal data can be separated. In reality, this boundary is not very distinctive since a point above the boundary can be normal, and a point falling below the threshold can be anomalous.

2. The notion of normal and anomalous behaviour can evolve as time progresses. For example, in financial fraud detection, the pattern of fraud can be dynamic since the adversaries adapt their strategies and behaviour to combat the detection technique (Fawcett & Provost 1997).
3. Labeled data is often unavailable, shifting the problem from supervised learning to unsupervised- or semi-supervised learning. This adds extra uncertainty to the model since there is no prior knowledge into which points are considered anomalous or not, and these points will instead have to be inferred from the data.

4. Noisy data can make distinguishing anomalies from normal data difficult.

When choosing a model for anomaly detection, the type of input data needs to be considered. Common types of input data are point data and time-series data. Point data is a simple data type where it is assumed to be no relationship among the data instances. Time series data is a sequence of data points collected in chronological order. As a consequence of the ordering of the data, the points are usually temporally correlated. Some examples of common attributes of the input data are discrete-, continuous- and categorical.

Another thing to consider when choosing an anomaly detection model is the kind of anomalies that will be detected. Anomalies are usually divided into three different types (Chandola et al. 2009). That is, point anomalies, contextual anomalies and collective anomalies. In figure 2.1, the different types of anomalies are represented in three different graphs.

![Figure 2.1 - Three different types of anomalies. A point anomaly is a data point considered abnormal compared to the rest of the dataset. A point that is anomalous when conditioned on a subset of adjacent points is called a contextual anomaly. When a collection of points are abnormal compared to the entire dataset they are denoted as a collective anomaly.](image)

A point anomaly, or outlier, is a data point that is considered abnormal compared to the rest of the dataset. Common methods for finding point anomalies are distance-based methods such as K-nearest neighbours, local outlier factor or probability-based models such as naïve Bayes classifier and Gaussian mixture model (Chandola et al. 2009).
Contextual (or conditional) anomalies refers to data that is considered anomalous when taking certain meta-information into account but is deemed normal otherwise. For instance, a 20 degrees Celsius outdoor temperature might not seem abnormal but with the context that it is December in northern Sweden it is anomalous. The context of the data is determined based on different attributes depending on what type of dataset is being used. For example, for spatial datasets, the location is the contextual attribute, whereas, for time series, time is the contextual attribute.

A collection of data points can be seen as anomalous in relation to the whole dataset, but when considering each point within the collection individually, they are not deemed anomalous. This type of anomaly is called a collective anomaly. An example is a stream of sensor data where the readings are alternating around the value zero. Suppose something unexpected happens to the sensor, like a loose cable effecting the connectivity. In that case, the readings might for a while be constant at zero and then eventually start to alternate again. A single value at zero is not abnormal in this setting. However, the collection of points that are all at zero can be considered anomalous.

2.2 Mathematical Background

This section gives a description of the mathematical theory behind the models used throughout the report.

First, the theory behind the Gaussian mixture model is laid out. It is a simple probabilistic method that does not take contextual information into account. The Gaussian mixture model will show how the other more complex methods compare to this simpler model.

Next, the mathematical concept of neural networks is explained. Including how the model is constructed, how it is used for classification and regression and how it is trained using backpropagation. The neural network is the backbone for the remaining models used in this research.

The recurrent neural network is introduced as an extension to the vanilla feedforward neural network. The recurrent neural network makes it possible to model sequential data with temporal dependencies. This section also introduces the recurrent neural network variant, LSTM. It will be used for different experiments.

In section 2.2.4, the concept of residual networks is discussed. It enables the training of deeper neural networks than is possible with vanilla Artificial Neural Network (ANN). This type of network is then used to derive neural ordinary differential equations. The section on neural ordinary differential
equations introduces two model variants that will be used for experiments. The LSTM-ODE and the Latent-ODE.

![Figure 2.2 – Roadmap showing relations between the different methods. The Gaussian mixture model is a standalone probabilistic model, whereas the rest of the models tested in this research are all based on neural networks. The arrows represent how each concept is derived. For example, neural-ODE is derived from residual networks which in turn is derived from the concept of neural networks. The red nodes indicate the Neural ODE-based methods used in this report.]

### 2.2.1 Gaussian Mixture Model

The Gaussian mixture model is a type of kernel method, used to model densities. The model represents the data as originating from sub-groups of data points within the total population. Each of these sub-groups, or components, is assumed to be normally distributed. However, no prior knowledge or assumption of which component the data points stems from is needed. The Probability Density Function (PDF) of the GMM has the following form:

$$f(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x | \mu_k, \Sigma_k),$$

(2.1)
where $\mathcal{N}(\cdot)$ is the PDF of a multivariate normal distribution with mean $\mu_k$ and covariance matrix $\Sigma_k$. $\pi_k$ is the mixture coefficients for each component $k$ such that $\pi_k \in [0, 1]$ and $\sum_{k=1}^{K} \pi_k = 1$ (Hastie et al. 2017).

The Expectation-Maximisation (EM) algorithm is commonly used to find a maximum-likelihood fit of the model parameters $\pi_k, \mu_k$ and $\Sigma_k$. The log-likelihood for $N$ data points is:

$$L(\{\pi_k, \mu_k, \Sigma_k\}_{k=1}^{K}) = \log f(X|\{\pi_k, \mu_k, \Sigma_k\}_{k=1}^{K}) = \sum_{i=1}^{N} \log \left( \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \right).$$

(2.2)

The EM algorithm is first initialised by the parameters $\{\pi_k^{(0)}, \mu_k^{(0)}, \Sigma_k^{(0)}\}_{k=1}^{K}$ (randomly sampled or deterministically defined), and then updated according to:

$$\mu_k^{(t+1)} = \frac{1}{N_k} \sum_{i=1}^{N} t_{ik}^{(t)} x_i,$$

$$\Sigma_k^{(t+1)} = \frac{1}{N_k} \sum_{i=1}^{N} t_{ik}^{(t)} (x_i - \mu_k^{(t+1)})(x_i - \mu_k^{(t+1)})^T,$$

$$\pi_k^{(t+1)} = \frac{1}{N} \sum_{i=1}^{N} t_{ik}^{(t)},$$

(2.3)

where $t_{ik}^{(t)}$ denotes the probability that $x_i$ is generated by the $k$-th Gaussian component using the current parameter values:

$$t_{ik}^{(t)} = \frac{\pi_k^{(t)} \mathcal{N}(x_i|\mu_k^{(t)}, \Sigma_k^{(t)})}{\sum_{j=1}^{K} \pi_j^{(t)} \mathcal{N}(x_i|\mu_j^{(t)}, \Sigma_j^{(t)})}.$$  

(2.4)

The quantity $N_k$ in equation (2.3) denotes the effective number of points in component $k$,

$$N_k = \sum_{i=1}^{N} t_{ik}^{(t)}.$$  

(2.5)

The number of components is usually determined using the Akaike information criterion (AIC) or the Bayesian information criterion (BIC). Both of the criteria give a trade of between the maximum log-likelihood estimation and the number of free parameters under the current model fit. They, therefore, can be used to compare performance of methods based on their fit while also penalising for too many free parameters and thereby preventing overfitting.
These criterions are obtained from

\[ \text{AIC} = -2\hat{\mathcal{L}} + 2P, \]
\[ \text{BIC} = -2\hat{\mathcal{L}} + \log(N)P, \] (2.6)

where \( P \) is the number of free parameters of the model

\[ P = Kd + \frac{1}{2}Kd(d+1) + K - 1. \] (2.7)

and \( \hat{\mathcal{L}} \) the maximised value of the log-likelihood function.

The BIC typically penalises the numbers of components more than the AIC. The number of components should be chosen to minimise either the BIC or the AIC (Eirola & Lendasse 2013).

### 2.2.2 Feedforward Neural Networks

An artificial neural network or ANN is a model where the input is mapped to the output by a series of computations. An ANN is usually represented by connected nodes (called neurons) that describe how the flow of information moves from input to output. The neurons are connected so the output from one node is the input to the next. The connections are weighted, making it possible to train the network to map the input to the output as accurately as possible. The neurons are often grouped into layers so the network consists of the following three components:

1. Input layer
2. Hidden layers
3. Output layer

In a feedforward neural network, input data \( \mathbf{x} \in \mathbb{R}^p \), where \( p \in \mathbb{N} \), is first passed from the input layer to the first hidden layer. Most of the computations are carried out within the hidden layers. The computations in layer \( i \) are denoted as \( f_i \). The output from one hidden layer is the input to the next layer. The output from the last hidden layer is passed to the output layer, where a final computation is carried out before the solution \( \hat{\mathbf{y}} \in \mathbb{R}^q \) (\( q \in \mathbb{N} \)) is obtained. The term depth of a neural network refers to the number of hidden layers. The dimensions of the hidden layers is referred to as the width of the network.
Figure 2.3 – A single Layer Perceptron. A network consisting of a three-dimensional input and a one-dimensional output connected with a single neuron/perceptron. Each of the inputs, \( x_i \), are multiplied with its corresponding weight, \( w_i \), and then added together with a bias term. This sum is then used as input to an activation function to produce the output \( y \).

The output from a hidden layer is often called a hidden state \( h \). The \( i \)-th hidden state (output of layer \( i \)), \( h_i \), is calculated in the following way

\[
h_i = f_i(h_{i-1}) = \sigma_i(b_i + W_i h_{i-1})
\]

(2.8)

Where \( b_i \) is a bias vector and \( W_i \) is a weight matrix. This means that the function \( f_i \) is composed of two separate functions. A linear function \( \ell_i(\cdot) \) and a non-linear function \( \sigma_i(\cdot) \). The function \( \sigma_i(\cdot) \) is usually referred to as an activation function. By adding these non-linearities, the model can learn non-linear mappings. As stated in the universal approximation theorem (Hornik et al. 1989), by having at least one hidden layer with a non-affine, continuously differentiable activation function and constrained width, the network can approximate any continuous function arbitrarily well.

There is a wide array of choices for an activation function \( \sigma(\cdot) \). Some of the most common activation functions can be found in table 2.1. The transformation in the output layer needs to be chosen so that the output falls within a desired range. For example, if the network is supposed to map a probability of the given input or do a classification task, the output function needs to be in the range \([0, 1]\), like a sigmoid function. If instead, the purpose of the network is to do a regression task, then a linear unit is suitable (Goodfellow et al. 2016, p. 181-191).
An ANN can be seen as a chain of functions where the output from one function is the input to the next. The mapping from the input to the output of a feedforward neural network of depth $d$ can be expressed by:

$$\hat{y} = f_{d+1}(f_d \cdots (f_2(f_1(x))))$$  \hspace{1cm} (2.9)$$

where $f_{d+1}$ is the output activation function, whereas the other $f_i$’s denote the intermediate hidden layer computations.

Figure 2.3 shows the simplest type of feedforward ANN, namely the single-layer perceptron. The network in the figure has three inputs, $x_1$, $x_2$ and $x_3$, that are passed to a single hidden node, where the inputs are multiplied with their corresponding weights and then summed together with the bias term. The result from this sum is then passed as input to the non-linear transformation. Finally, the result from the activation function gives the desired output. The term feedforward comes from the fact that the information only flows in one direction through the network, from the input to the output. Figure 2.4 illustrates a more complicated network architecture containing two hidden layers with four neurons in each layer. The network has six inputs and two outputs.
Figure 2.4 – An illustration of a fully connected multi-layer perceptron containing two hidden layers. The network has a six-dimensional input and a two-dimensional output. The width of the hidden layers is four.

Backpropagation

A common way to train neural networks is using a gradient descent based algorithm. Local minima of some loss or cost function $C$ is found by iteratively updating the weights and biases in the opposite direction of the gradient of $C$. The most common approach to numerically compute the necessary gradients is the backpropagation algorithm (Rumelhart et al. 1986). The idea is to use the chain rule to obtain the derivative of the desired weights or biases in the composite function in equation (2.9). To give a simple example of the chain rule for a composite function consisting of two functions that take in a vector-valued input, consider $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$ and the functions $g(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^n$ and $f(\mathbf{y}) : \mathbb{R}^n \to \mathbb{R}$, where

$$z = f(g(\mathbf{x})). \tag{2.10}$$

Then the derivative of $z$ with respect to an element $x_i$ in the vector $\mathbf{x}$ is

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \tag{2.11}$$

The weight that connects neuron $k$ in layer $(\ell - 1)$ to neuron $j$ in layer $\ell$ is denoted by $w_{\ell}^{j,k}$ and the bias neuron $j$ in layer $\ell$ is denoted by $b_\ell^j$. The intermediate quantity $z_\ell$, the input to the $\ell$-th activation function, is defined as $z_\ell \equiv W_\ell h_{\ell-1} + b_\ell$. The quantity $\delta_\ell$ gives a relation how the cost function
$C$ changes with respect to a change in the intermediate quantity $z_\ell$. $\delta^j_\ell$ can be seen as the error of neuron $j$ in layer $\ell$ and is defined by $\delta^j_\ell \equiv \frac{\partial C}{\partial z^j_\ell}$.

The backpropagation algorithm for a feedforward network of depth $L$ follows the following steps (Goodfellow et al. (2016), p. 204-221, Nielsen (2015), ch. 2):

1. **Input:** Compute the first hidden state $h_1 = \sigma_1(W_1 x + b_1)$.

2. **Feedforward:** For each $\ell = 2, 3, ..., L$ compute $z_\ell = W_\ell h_{\ell-1} + b_\ell$ and $h_\ell = \sigma_\ell(z_\ell)$.

3. **Output error:** compute the output error vector $\delta_L = \nabla h C \circ \sigma'_L(z_L)$ where $\circ$ denotes the element-wise product between the two vectors.

4. **Backpropagate** the error through the network.
   For $\ell = L - 1, L - 2, ..., 2$, compute $\delta_\ell = ((W_{\ell+1})^T \delta_{\ell+1}) \circ \sigma'_\ell(z_\ell)$.

5. **Gradients:** The desired gradients w.r.t. the weights and biases can be obtained from $\frac{\partial C}{\partial w^T} = h^{T}_{\ell-1} \delta^j_\ell$ and $\frac{\partial C}{\partial b^T} = \delta^j_\ell$.

This gradient can then be used to update the desired weight iteratively. An update for the $n$:th iteration is expressed by

$$w_n = w_{n-1} - \eta \frac{\partial C}{\partial w_{n-1}},$$

where $\eta > 0$ is the learning rate (subscripts and superscripts specifying the layer and neurons are omitted from equation (2.12) for brevity).

### 2.2.3 Recurrent Neural Networks

Vanilla neural networks cannot utilise the context within sequential data. Given a time-series as a vector input, the method handles each point independently from each other.

RNNs were introduced to handle sequential data. To do so, they are built to "remember" previous inputs. Following is a short introduction to RNNs (Goodfellow et al. 2016, p. 378-394).

As our problem now involves sequential data, let $x_i \in \mathbb{R}^p$, where $p \in \mathbb{N}$, be input $i$ of the sequence. Similarly, let $y_j \in \mathbb{R}^q$, where $q \in \mathbb{N}$, be output $j$ of the RNN. The number of inputs and outputs varies between problems. The different configurations can be seen in figure 2.5.
The configurations of interest for this thesis concerning multivariate time-series anomaly detection are the one-to-many and many-to-one cases. A many-to-one configuration can take several subsequent time points as input and predict the next value in the sequence. Similarly, a one-to-many model can predict a time-series forward in time from a given initial value.

To model memory within the method, a hidden state $\mathbf{h}$ is introduced. It is a vector of arbitrary dimension $d$ unrelated to the input and output dimensions. The hidden state is updated every time a new input is provided so $\mathbf{h}_t$ is the value of the hidden state at time $t$. The mechanics are then as follows.

Before any input is provided at time zero the hidden state $\mathbf{h}_0$ is the zero vector as there is nothing to remember. For every input $\mathbf{x}_i$ from the sequence, the hidden state is updated according to the following equation:

$$
\mathbf{h}_i = f(W_x \mathbf{x}_i + W_h \mathbf{h}_{i-1} + \mathbf{b}_h),
$$

(2.13)

where $f$ is the activation function, $W_h$ and $W_x$ the weight matrices for $\mathbf{h}$ and $\mathbf{x}$ respectively and $\mathbf{b}_h$ the bias when updating $\mathbf{h}$. That is, the newest input is combined with the old hidden state to create the new hidden state. From every hidden state $\mathbf{h}$ an output can be generated using the following equation:

$$
\mathbf{y}_i = f(W_y \mathbf{h}_i + \mathbf{b}_y),
$$

(2.14)

where $f$ is the activation function, $W_y$ is the weight matrix and $\mathbf{b}_y$ the bias for the output calculation. Figure 2.6 shows the inner working of a single RNN.
cell.

Figure 2.6 – The architecture of an RNN cell. The new input $x_i$ and old hidden state $h_{i-1}$ are combined to create a new hidden state $h_i$. An output $y_i$ can be generated from this new hidden state.

The training of an RNN is similar to a vanilla ANN. The difference is that many inputs influence a single output. Therefore, the backpropagation algorithm is modified into the backpropagation through time algorithm for RNNs. It will not be covered here, but the interested reader can find it in Goodfellow et al. (2016), p. 384-386. The general idea is to feedforward through the network for each input in the sequence. The backpropagation then has to propagate back through the network and through time as the effect of each input of the sequence is evaluated.

While RNNs have their benefits in handling sequential data and being flexible when it comes to the number of inputs and outputs, they do have their flaws. First off is the vanishing gradient problem. The gradient used to compute the weight update can get close to zero, preventing the network from learning. This problem gets worse as the network becomes deeper. Secondly, the similar problem of exploding gradients where instead of tending to zero, the gradients tend to infinity. Lastly, an RNN is not well suited for irregularly sampled data. The index $i$ denotes the numbering of the input within the input sequence. For a sequence of time-series data, the index can be interpreted as time. The problem arises when the time series is irregularly sampled as the update step within RNNs treats each update step as being of uniform length. Inserting an irregularly sampled time series into an RNN will hinder the model’s ability to learn the underlying dynamics.
Different variations of RNNs have been introduced to tackle these problems. The most notable ones were introduced to tackle the vanishing/exploding gradient problems, the LSTM and Gated recurrent unit (GRU). Within this thesis, the focus will be on LSTMs.

Long Short-Term Memory

The idea behind LSTMs is to further expand on the concept of memory within the model (Hochreiter & Schmidhuber 1997). While remembering previous inputs is beneficial, it can be helpful to be able to forget them as well. In addition to a hidden state, LSTMs have a cell state \( c \) which keeps track of which of the previously seen inputs in the sequence are still of value and should be kept in memory. This cell state, therefore, adds the ability to forget. The cell state at time \( t \) is denoted as \( c_t \) and is initialised as zero like the hidden state.

Even though the conceptual change between RNNs is small, the calculations become more complicated. The inner workings of the method for each step are as follows. We start with an input \( x_i \), the previous hidden state \( h_{i-1} \) and the previous cell state \( c_{i-1} \). First off, we calculate two vectors \( f_t \) and \( r_t \). Both are vectors that contain values between zero and one. The values can be interpreted as to what extent the corresponding information should be remembered or forgotten: zero representing forgetting completely and one remember completely. \( f \) will be applied to the old cell state as it is being forgotten to some extent and \( r \) to the calculated update values for \( c \) as they are being added to memory or remembered. The following equations show how they are calculated

\[
f_i = \sigma(W_{h,f}h_{i-1} + W_{x,f}x_i + b_f), \quad (2.15)
\]

\[
r_i = \sigma(W_{h,r}h_{i-1} + W_{x,r}x_i + b_r), \quad (2.16)
\]

where \( W_{u,s} \) is the weight matrix for input \( u \) in this step \( s \) and \( b_s \) the bias in the step. The steps being either \( f \) or \( r \). Having calculated how much to forget and remember, the next step is to calculate the new cell state using the following equation

\[
c_t = f_t \circ c_{i-1} + r_t \circ \tanh(W_{h,c}h_{i-1} + W_{x,c}x_i + b_c), \quad (2.17)
\]

where \( \circ \) denotes element-wise multiplication. All that remains is to update the hidden state \( h \) with the following equation

\[
h_i = \tanh(c_i) \circ \sigma(W_{h,o}h_{i-1} + W_{x,o}x_i + b_o). \quad (2.18)
\]
Outputs are then created like in vanilla RNNs from $h_i$ and equation (2.14). Figure 2.7 shows the inner architecture of a single LSTM step.

Figure 2.7 – The architecture of an LSTM cell. The figure shows first how the new input $x_i$ and old hidden state $h_{i-1}$ are combined with the old cell state $c_{i-1}$ to calculate how much of the cell state should be "forgotten". Then combined to calculate how much should be added to the cell state or "remembered". Finally, the new cell state $c_i$ is combined with the old hidden state and the new input to create the new hidden state $h_i$. An output $y_i$ can be generated from this new hidden state.

One might wonder how this fixes the vanishing gradient problem. The key is the addition step in equation (2.17). Instead of the updates being the product of successive multiplications where the error can quickly explode or vanish, they are the product of successive additions which allows information to flow further within the LSTM without exploding or vanishing.

### 2.2.4 Residual Networks

A Deep Neural Network (DNN) is an ANN consisting of many layers. By adding more layers to the network, the model can integrate more levels of features and thus represent more complex functions (Goodfellow et al. 2016, p. 166). Therefore, adding more layers should lower the training and test error (to some extent). Unfortunately, in practice, this is not always the case. This is mainly due to two problems: the vanishing/exploding gradient problem and the degradation problem.

The vanishing/exploding gradient problem arises in DNNs because the gradients in the early layers of the network depend on the gradients in the subsequent layers. Meaning that the gradients are computed by multiplying derivatives. If these derivatives become very small or large, the gradients can vanish or explode. This hinders the model’s training, preventing it from reaching a local minimum. This problem has been partially solved by
normalised weight initialisation (Glorot & Bengio 2010) and by normalising the intermediate layer inputs (batch normalisation) (Ioffe & Szegedy 2015).

When increasing the depth of a network, it has been observed that the accuracy gets saturated and then rapidly degrades. That is, first, the accuracy increases as layers are added to the network until it reaches a point where the accuracy starts to decrease again as the model gets deeper. The test and training error increase for deeper models, indicating that the degradation is not caused by overfitting. Instead, it comes from the fact that more complex networks are harder to optimise (train). He et al. (2016a) introduced a way to circumvent the degradation of training accuracy by adding skip connections consisting of identity mappings between the layers. Figure 2.8 shows the building block of this type of network. In this example, the skip connection jumps over one weight layer and an activation function, but it can skip an arbitrary number of layers. A very deep network can be created by stacking several of these so-called residual blocks together. These types of networks are called Residual Networks (ResNets). A simple explanation from He et al. (2016a) as to why this modification helps is as follows: Given a shallow network, one can create a deeper network that is guaranteed to perform no worse than the shallower counterpart by adding layers that perform identity mappings. The ResNet architecture is initialised with identity mappings as its defaults. That means that the network does not have to use all the layers it has and given no modifications, the remaining residual blocks perform identity mappings. Contrasting that to ANNs, where the unnecessary layers are initialised as near-zero mappings and would have to learn an identity mapping to get the same result.

Figure 2.8 – A simple residual block illustrating the identity mapping as a skip connection that skips one weight layer. These residual blocks are the building blocks for residual networks. Introducing identity mappings between layers enables training very deep neural networks without encountering degradation of training accuracy.
The transition between intermediate hidden states in a ResNet can be represented as (He et al. 2016b)

$$h_{i+1} = h_i + f(h_i, W_i).$$  \hfill (2.19)

If all of the residual blocks that make up a ResNet of depth $L$ have the same architecture, we obtain the following expression for the mapping between the first and the last hidden state:

$$h_L = h_0 + \sum_{i=0}^{L-1} f(h_i, W_i).$$  \hfill (2.20)

### 2.2.5 Neural Ordinary Differential Equations

In the state update equation for residual networks (see equation (2.19)), the transition from one state to another is done by adding an incremental change to the current state. This resembles an Euler discretisation with step size one of some continuous function (Haber & Ruthotto 2018, Chang et al. 2018, Weinan 2017). In the limit of the step size (which corresponds to the number of layers $L \to \infty$), the derivative of the hidden states can be parameterised using an ODE specified by a neural network to obtain the so-called Neural ODE (Chen et al. 2019). This Neural ODE corresponds to the following initial value problem:

$$\lim_{\Delta t \to 0} \frac{h_{t+\Delta t} - h_t}{\Delta t} = \frac{dh(t)}{dt} = f(h(t), t, W),$$  \hfill (2.21)

where $h(0)$ is the initial value of the ODE. Thus, Neural ODEs can be seen as the deep limit of ResNets. The solution to equation (2.21) at some time $T$, is said to give the desired final hidden state $h(T)$. Hence, an expression for the mapping between the input and the output hidden state is given by

$$h(T) = h(0) + \int_0^T f(h(t), t, W)dt.$$  \hfill (2.22)

The desired output of the model, $\hat{y}$, can be obtained either directly from the final hidden state of the Neural ODE, $h(T)$, or by inserting the final state into an output layer $h_y(h(T))$. The main reason for adding an output layer is to ensure that the output gets the desired dimension of $\hat{y}$ (Massaroli et al. 2020).

The initial value problem of equation (2.21) can be solved using a numerical differential equation solver. A variety of known network types
can be seen as different fixed step numerical integration schemes of a neural ordinary differential equation (see table 2.2.)

<table>
<thead>
<tr>
<th>Type of network</th>
<th>Fixed step integration scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet, RevNet</td>
<td>Forward Euler</td>
</tr>
<tr>
<td>PolyNet</td>
<td>Backward Euler</td>
</tr>
<tr>
<td>FractalNet</td>
<td>Runge-Kutta</td>
</tr>
<tr>
<td>DenseNet</td>
<td>Runge-Kutta</td>
</tr>
</tbody>
</table>

Table 2.2 – Relationship between some ANNs and integration scheme for Neural ODEs (Chang et al. 2018, Lu et al. 2018).

Many ODE-solvers use adaptive step sizes to solve to an initial value problem within a given error tolerance. This makes it possible to have an explicit trade-off between the model’s accuracy and the computational cost. In order for the solution to equation (2.22) to be unique, the non-linear activation functions within the dynamic \( f(h(t), t, W) \) must be Lipschitz continuous (Chen et al. 2019), such as tanh, sigmoid or ReLu (see table 2.1).

The concept of depth for these continuous neural networks is different from that of discrete layered networks. In the setting of Neural ODE, the depth is determined by the Number of Function Evaluations (NFE) of the dynamics function \( f \) in a forward pass. Chen et al. (2019) verified that the error decreases with NFE and the computation time increases linearly with respect to NFE. The NFE for each forward pass also increases the longer the model is trained. Likely because the ODE-solver is adapting to the increasing complexity of the model.

A drawback of ODEs is that the trajectories in the vector fields cannot cross each other. This is due to the existence and uniqueness theorem for autonomous ordinary differential equations (Zill 2018, p. 36). As a consequence of this, the Neural ODE can only model bijective transformations. Figure 2.9b shows that a Neural ODE fails to map the input -1 and 1 to the output 1 and -1, respectively, as the trajectories would have to cross. For the trajectories to cross successfully, a point in the vector field would have two different solutions which is not possible. In contrast, the ResNet in figure 2.9a managed to map them successfully. However, the problem can be circumvented by input augmentation. One way is to augment the input space by concatenating a null vector (zero vector) to the input vector, which increases the dimension (Dupont et al. 2019). The input \( h(0) \) to the Neural ODE instead becomes \( h(0) = [x, 0]^T \). Another approach is to couple the Neural ODE with an input neural network, making the initial condition of
the ODE-solver instead become $h(0) = h_x(x)$ (Massaroli et al. 2020). The dimension of the output from the input network $h_x$ needs to be larger than that of the input ($\text{dim}(h_x(x)) > \text{dim}(x)$). The downside of this is that extra parameters are introduced compared to the first augmentation strategy.

Figure 2.9 – Trajectories of ResNet (left) and Neural ODE (right). Inputs are to the left and outputs to the right. The models were trained to map input 1 to -1 and input -1 to 1. ResNets can be seen as a discretised ODE using forward Euler with a discretisation step size of one. Since ResNets take discrete steps, they can have crossing trajectories, whereas Neural ODEs cannot.

To train a Neural ODE, one can keep track of all the intermediate quantities in a forward pass and then directly backpropagate through the internal operations of the ODE-solver. However, this imposes extra memory cost. Another alternative is to use the adjoint sensitivity method, which is a method that was introduced by Pontryagin (1962) for sensitivity analysis within optimisation using reverse mode differentiation. Chen et al. (2019) derived a way to use the adjoint method as a continuous backpropagation scheme to train Neural ODEs. The aim of the training phase is to minimise some loss function $L$ with respect to the weights $W$. This loss function is different from the discrete layered network counterpart in the sense that the loss depends on the entire trajectory from the input to the output. The loss at the output of the Neural ODE network is

$$L(h(T)) = L\left(h(0) + \int_0^T f(h(t), t, W)dt\right)$$

$$= L(\text{ODESolve}(h(0), f, t_0 = 0, t_1 = T, W)),$$

where the ODE-solver is treated as a black box. To update the weights according to some gradient descent based algorithm, the gradients of $L$ w.r.t.
\( W \) is required. A quantity \( a(t) \) called the adjoint state is introduced, which relates how the gradients of the loss depend on the continuous hidden state \( h(t) \)

\[
a(t) = \frac{\partial L}{\partial h(t)}.
\]

(2.24)

From this, Chen et al. (2019) showed that the trajectory of the adjoint can be determined by a second augmented ODE (derived from the chain rule)

\[
\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(h(t), t, W)}{\partial h(t)}.
\]

(2.25)

By starting at the final point in time \( T \), equation (2.25) can be integrated backwards in time to any arbitrary time point \( t \) to obtain \( a(t) \). Given the initial condition \( a(T) = \frac{\partial L}{\partial h(T)} \), which can be obtained from the result of the forward pass, we have that

\[
a(t) = \frac{\partial L}{\partial h(t)} = a(T) + \int_T^t a(t')^T \frac{\partial f(h(t'), t, W)}{\partial h(t)} dt'.
\]

(2.26)

Under the assumption that the weights are constant w.r.t. time (\( \frac{dW}{dt} = 0 \)), the desired gradients of the loss w.r.t. the weights can be obtained by

\[
\frac{dL}{dW} = -\int_T^0 a(t)^T \frac{\partial f(h(t), t, W)}{\partial W} dt.
\]

(2.27)

Thus, the adjoint sensitivity method has constant memory cost, since no intermediate quantities need to be stored from each forward pass (Chen et al. 2019). A drawback is that the trajectory of \( h(t) \) needs to be recomputed in the backwards pass, introducing extra numerical error and adding to the computational complexity.

**RNN-ODE**

Many different modifications to RNNs have been developed to deal with irregularly sampled time series. The simplest way is to modify the data using averages or imputations to add the missing datapoints making the time series regularly sampled. By doing so we are possibly losing valuable data or adding data points that are not true to the underlying system. A better approach can be to construct a continuous-time model with a latent state defined for all times. Rubanova et al. (2019) modified RNNs to apply them to irregularly sampled time series. The hidden state is modelled using a Neural ODE. That is, the
state between observations is a solution to an ODE

\[ h'_i = \text{ODESolve}(f_\theta, h_{i-1}, (t_{i-1}, t_i)). \] (2.28)

Given a new observation, the hidden state is updated like in equation (2.13), with the only difference being \( h_{i-1} \) is replaced with \( h'_i \). Figure 2.10 shows how this alters the architecture of the RNN.

Rubanova et al. (2019) showed that this RNN variant could outperform other RNN variants when the data is sparse and has the aforementioned benefit of seamlessly incorporating irregularly sampled time-series data. A further extension can be made to this method to create ODE-LSTMs. Lechner & Hasani (2020) introduced this method where the hidden state \( h \) of an LSTM is again updated using a Neural ODE. The cell state \( c \) is not affected by this change. They showed that ODE-LSTMs outperformed ODE-RNNs in a variety of tasks. For this thesis we will work with ODE-LSTMs.
Latent-ODE

In the original Neural ODE paper, the authors describe how to implement a generative latent function time-series model using a Neural ODE (Chen et al. 2019). The same authors further improved upon that method in a later paper by introducing Latent-ODEs (Rubanova et al. 2019). A flaw in using vanilla Neural ODE in time-series prediction is that they are completely determined by the initial state. Latent-ODEs add a latent-variable time series model where we want the initial latent state $z_0$ to contain information on the history of the true time series up until that point. Using a Neural ODE to predict that latent time-series forward in time means we can use all the gathered observations for the prediction, not just the first observation. Mathematically we have:

$$z_0 \sim p(z_0) \quad (2.29)$$
$$z_0, z_1, \ldots, z_N = \text{ODESolve}(f_\theta, z_0, (t_0, t_1, \ldots, t_N)) \quad (2.30)$$
$$x_i \sim p(x_i|z_i), \quad i = 1, 2, \ldots, N \quad (2.31)$$

The framework used for the training and prediction is a Variational autoencoder (VAE) (Kingma & Welling 2014). A VAE is a combination of an encoder and a decoder. The encoder takes in observations and learns a fixed-dimensional embedding of the data. The decoder then learns to reconstruct the original data from the embedding. For that, we need to estimate the approximate posterior $q(z_0|\{x_i, t_i\}_{i=0}^N)$. That is, the probability distribution of the initial latent state given the input data. We will use the ODE-LSTM introduced above to get the mean and standard deviation of the approximate posterior:

$$\mu_{z_0}, \sigma_{z_0} = g_e(\text{ODE-LSTM}(\{x_i, t_i\}_{i=0}^N)), \quad (2.32)$$
$$q(z_0|\{x_i, t_i\}_{i=0}^N) = \mathcal{N}(\mu_{z_0}, \sigma_{z_0}), \quad (2.33)$$

where $g_e$ translates the final hidden state of the ODE-LSTM at time $t_0$ to the mean and standard deviation. We can then sample $z_0$ from the posterior to get the initial latent state. Using a Neural ODE to calculate the dynamics forward in time and another small neural network $g_d$ to translate the latent variable back to the true observation space at each time of interest, one can reconstruct the original input. Here, $g_e$ and $g_d$ are small single layer ANNs which make sure the outputs are of the correct dimension. The dynamics of the final model can be seen in figure 2.11.
Figure 2.11 – The dynamics of a latent-ODE model. First, the sequence is inputted backwards into an ODE-LSTM which encodes the information into the mean and standard deviation of the initial latent state $z_0$. The latent variable is then calculated forward in time using a Neural ODE. The reconstructed points can be recovered from the latent space by decoding them into the true observation space.

The objective function we wish to minimise during training is the negative evidence lower bound (ELBO), which is of the form:

$$\text{ELBO}(\theta, \phi) = \mathbb{E}_{z_0 \sim q_\phi(\cdot)} \left[ \log p_\theta(x_0, \ldots, x_N) \right] - KL[q_\phi(\cdot) || p(z_0)].$$  \hspace{1cm} (2.34)

The first term is the reconstruction loss and can be interpreted as the average agreement between the predictions and the true data. The second term is the Kullback–Leibler divergence. It is a measure of the difference between two probability distributions. In our case the approximate posterior $q_\phi$ and the prior we assign to the first latent state. Most often, the prior is a standard normal distribution. This term keeps the encoder from becoming deterministic and returning a point distribution and keeps the standard deviation low. Hence, the encoder only encodes the most important information in the latent state. The architecture can be seen in figure 2.12.
Figure 2.12 – The architecture of a variational autoencoder. The input is encoded into the mean and variance for the initial latent state. The initial latent state is then sampled and decoded back to the original space. The loss of the model is the ELBO loss which is a combination of the reconstruction loss and the Kullback-Leibler divergence.
Chapter 3

Methods

This chapter describes the methods used to answer the research question. That includes the models, datasets, metrics and experimental settings. First, in Section 3.1, a description of how the models introduced in Chapter 2 will be used for anomaly detection. The metrics used to compare the performance of the models will then be covered in Section 3.2, along with the plot layout used for comparisons. Secondly, in Section 3.3, details will be given about the datasets used and the preprocessing of that data. A description of the experiments done on said datasets is included. Lastly, in Section 3.4 and 3.5, the models used and the experimental settings for each model will be outlined.

3.1 Methods to detect anomalies

The method most suitable for anomaly detection depends upon the problem at hand. The best anomaly detector in time-series is not necessarily well suited to anomaly detection problems in images or medical records. Chandola et al. (2009) name different strategies that have been used in the past for anomaly detection. Strategies include both parametric and non-parametric statistical methods, neural networks, spectral methods and clustering methods, to name a few. In our case, we will be looking at a parametric statistical method and neural network based methods specifically for time-series data from physical sensors.

One goal of anomaly detection is to detect surprising never-before-seen behaviour in the data. Therefore, this behaviour in the data is not available in the data at training time and in practice, supervised learning is often not possible. Using semi-supervised or unsupervised learning can be more appropriate. To use semi-supervised learning, one can gather data under
controlled conditions and make sure that there are no anomalies present. The data collected is labelled as "normal", and models can be trained to reconstruct or predict the data forward in time. When test data is fed to a model trained on this normal data, bad predictions can imply anomalous behaviour. It can be difficult to do semi-supervised learning successfully. The training data ("normal data") gathered needs to be sufficiently representative of all possible normal behaviours. Otherwise, normal instances from never-before-seen behaviours will likely be classified as anomalous. While it is hard to include all normal behaviours, it is also difficult to exclude all anomalies. The monitored asset can function normally while having degrading performance, which can only be seen in the data. A complete breakdown only happens after a prolonged period of this degrading run. Therefore, having confidence in the normally labelled data for semi-supervised learning can be treacherous.

When it is impossible to control conditions sufficiently well to exclude anomalies, one can use unsupervised learning. Then the assumption is made that normal instances are far more frequent than anomalies in the data and that training a model on this data will create a model robust to any outliers (Chandola et al. 2009). That is, assume the model will better predict or reconstruct the normal instances because they are more common than the anomalous ones. If test data results in bad predictions of the model, it is classified as anomalous.

The semi-supervised and unsupervised approaches are very similar in this setting as can be seen above. The difference lies in the assumption made of the training data. Whether it is assumed only to contain normal instances or assumed to contain a majority of normal instances where the anomalous minority will not affect the training of models. For this thesis, we will be working in a semi-supervised setting. The training data will not contain any anomalous points, and the test data will contain a mixture of anomalous and normal points. The methods used will also work in an unsupervised setting, but the results below will not cover how robust each method is to anomalies in the training data.

3.1.1 Anomaly score

An anomaly detection model will classify data points as either anomalous or normal based on an anomaly score, which differs between models. A well-performing anomaly detector will assign anomaly scores so that the normal and abnormal points can be separated.

In this thesis, we compare three different model types for anomaly
Methods

31
detection, namely, autoregressive models, variational autoencoders and a statistical model. An autoregressive model is a model that predicts future values based on the current and past values from the same time-series. A common anomaly score for autoregressive models is the Residual Loss (RL). The residual loss measures the dissimilarity between a prediction $\hat{x}_t$ and its true value $x_t$, the RL is obtained from

$$RL = |\hat{x}_t - x_t|.$$  

(3.1)

A variational autoencoder is a model that tries to reconstruct the input by sampling from a fixed-dimensional embedding of the original input. A commonly used anomaly score for variational autoencoders is the reconstruction loss (see equation (3.2)). The reconstruction loss measures the expected log-likelihood of the true data under the approximate posterior over the latent variables.

$$\text{Reconstruction loss} = \mathbb{E}_{\mathbf{z}_0 \sim q_\phi(\cdot)}[\log p_\theta(x_0, \ldots, x_N)]$$  

(3.2)

Statistical models attempt to learn the distribution of the data based on assumptions regarding, e.g. the parametric family or feature correlation. The anomaly scoring technique for statistical models like the Gaussian Mixture Model can be the log-likelihood of the data point of interest. If the probability that the data point belongs to the mixture of Gaussian components is small, it can be an indicator that the point is anomalous. Another popular scoring technique for GMMs is the Mahalanobis distance $D_M$, which gives a distance measure between the point and the Gaussian component $k$ that the point is predicted to belong to,

$$D_M^{(k)}(\mathbf{x}) = \sqrt{(\mathbf{x} - \mathbf{\mu}_k)^\top \Sigma_k (\mathbf{x} - \mathbf{\mu}_k)}.$$  

(3.3)

A threshold value needs to be specified to classify a point as anomalous based on an anomaly score. This threshold will act as a boundary between the normal and abnormal regions. If the anomaly score for a point exceeds some threshold value, the point will be labelled as anomalous. A poor choice of a threshold boundary will classify too many or too few points as anomalous. Therefore, finding a suitable threshold value is critical to ensure that the anomaly identifier correctly classifies the points. For semi-supervised and unsupervised anomaly detection models, a common way to determine the threshold value is to pick a percentile of the anomaly score from the training data. For example, anomaly scores in the test data higher than 99% of the
anomaly scores for the training data will be classified as anomalous. Another way to choose a cutoff value is make a histogram of the anomaly scores and check if there are distinctive peaks that separate the anomaly score of the normal and abnormal data.

### 3.2 Benchmarking

When comparing the performance of different models that conduct classification, accuracy is a popular metric. However, in the anomaly detection framework, this metric can be deceiving. Often, only a small fraction of the points in the test data are labelled as anomalous. This means that a model labelling *all* the points as normal will receive a very high accuracy score when in reality, it has not detected a single anomaly.

To benchmark the different anomaly detection methods, some different metrics that compare the predictions to the ground truth will be used. These metrics require the test data to be labelled to compute the various metrics. If there is no labelled data available, one can manually label the points. When comparing the classifications obtained from the anomaly detector to the true labels, it is possible to divide the classified points into four different groups, they are:

1. **True Positive (TP):** Point correctly classified as anomalous
2. **False Positive (FP):** Point incorrectly classified as anomalous
3. **True Negative (TN):** Point correctly classified as normal
4. **False Negative (FN):** Point incorrectly classified as normal

#### 3.2.1 Receiver Operating Characteristics

A way to compare the model’s performance without explicitly deciding a threshold value for the anomaly score is to use Receiver Operating Characteristics (ROC) curves. The ROC curve gives a visual representation of the trade-off between the true positive rate (TPR)

\[
TPR = \frac{TP}{TP + FN} \tag{3.4}
\]

and the false positive rate (FPR)

\[
FPR = \frac{FP}{FP + TN}. \tag{3.5}
\]
The TPR is a measure of the percentage of points correctly flagged as anomalous, and the FPR is a measure of the percentage of points incorrectly classified as anomalous. The ROC curve is obtained by plotting the TPR on the y-axis and the FPR on the x-axis for increasing threshold values. Thus the ROC curve can be seen as a metric of how well the anomaly detector manages to separate the anomaly score of the normal and anomalous data.

Figure 3.1 – Example of Receiver Operator Characteristic (ROC) curves for detectors of varying performance (to the left) with corresponding histograms of the anomaly score (to the right). The true positive rate measures the percentage of points correctly flagged as anomalous and the false positive rate measures the percentage of points incorrectly classified as anomalous. The ROC curve is obtained by plotting the true positive rate on the y-axis and the false positive rate on the x-axis for increasing threshold values. The area under the curve is denoted as AUC. The optimal value of the AUC is one. The three models shown are a perfect model which manages to separate the classes perfectly. A random model which assigns a random anomaly score to each instance, and no separation of classes is possible. Finally, an average model or a model that one might commonly see in a real-world setting. The average model manages to separate the classes into two different distributions with overlap, which gives an AUC score between 0.5 and one.

For a perfect detector with a separable cutoff value for the anomaly score, the TPR would be equal to one, and the FPR would be equal to zero. This means that the anomaly detector managed to separate the anomaly score of the anomalous and normal points, as shown in the histogram of the anomaly score for the perfect detector in figure 3.1. However, in practice, there is a trade-off
between the two since there will generally be an overlap of the anomaly scores for the normal and anomalous points. A higher TPR will mostly result in a higher FPR. The average detector in figure 3.1 managed to create an overlap between the two classes, but there is still possible to find a some what suitable threshold value that can act as a boundary. A random detector will produce a completely overlapping anomaly scores for the two classes. This would mean that it is impossible to find a suitable threshold value. The area under the curve (AUC) of the ROC is frequently used as a metric to compare different models. The AUC can be regarded as the model’s ability to distinguish between the normal and anomalous data. The range of values of the AUC is between zero and one. In general, the higher the value of AUC, the better (Fawcett 2006). While zero is the lowest value, it is not the worst as it corresponds to a classifier that is wrong 100% of the time. Therefore assigning the opposite class to its prediction results in a perfect classifier. Therefore, the worst AUC score is 0.5 as it corresponds to the random classifier mentioned above. For the random classifier, every choice of a threshold leads to equally many normal and anomalous points, on average, on each side of the threshold.

3.3 Data

The dataset used for evaluating the anomaly detection models is the WISDM Human Activity Recognition (HAR) dataset. It is open source and can be obtained from the link www.cis.fordham.edu/wisdm/dataset.php (link last checked 2021-11-09). This section describes the dataset, how it is used for anomaly detection and data preprocessing. Experiments were also done on the Skoltech anomaly benchmark dataset (SKAB) (Katser & Kozitsin 2020). The results did not give conclusive results and are therefore not included in the main report. The results and discussion on that dataset are included in appendix A.

3.3.1 Human Activity Recognition

The human activity recognition dataset is released by the Wireless Sensor Data Mining (WISDM) Lab (Kwapisz et al. 2011). The dataset contains 1,098,207 time-series data points obtained from 36 different participants performing different activities. The data was collected at a rate of 20Hz (1 sample every 50ms) using android phones, and each data point contains information about six different features. There were some failed samples, as can happen with real-world sensor data, meaning that there were some irregularities in the
timesteps between samples. However, 90% of the timesteps were 50ms, so the dataset is mostly regularly sampled. Table 3.1 shows the data type, unit and outcome range.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Data type</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>Nominal (integer)</td>
<td>-</td>
</tr>
<tr>
<td>activity</td>
<td>Nominal (string)</td>
<td>-</td>
</tr>
<tr>
<td>timestamp</td>
<td>Numeric (integer)</td>
<td>ns</td>
</tr>
<tr>
<td>x-acceleration</td>
<td>Numeric (floating point)</td>
<td>m/s²</td>
</tr>
<tr>
<td>y-acceleration</td>
<td>Numeric (floating point)</td>
<td>m/s²</td>
</tr>
<tr>
<td>z-acceleration</td>
<td>Numeric (floating point)</td>
<td>m/s²</td>
</tr>
</tbody>
</table>

Table 3.1 – Features of the Human Activity Recognition dataset.

The timestamp shows the relative uptime of the phone, i.e., the time since the mobile device was last turned on (in nanoseconds). Worth noting is that the gravitation is included in the accelerometer recordings, so if the phone is at rest, the accelerometers will not necessarily be equal to zero. There are six different activities the users performed. The activities together with their corresponding relative frequencies can be found in the following table:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Relative frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walking</td>
<td>38.6%</td>
</tr>
<tr>
<td>Jogging</td>
<td>31.2%</td>
</tr>
<tr>
<td>Upstairs</td>
<td>11.2%</td>
</tr>
<tr>
<td>Downstairs</td>
<td>9.1%</td>
</tr>
<tr>
<td>Sitting</td>
<td>5.5%</td>
</tr>
<tr>
<td>Standing</td>
<td>4.4%</td>
</tr>
</tbody>
</table>

Table 3.2 – Activities of the Human Activity Recognition dataset.

An illustration of a series of accelerometer readings and their corresponding activities is shown in figure 3.2.
To use the dataset for anomaly detection, one or several of the activities can be chosen as the anomalous class. Training is then done on data where the anomalous activity/activities are excluded. This makes it a semi-supervised anomaly detection problem. For the experiments throughout this project, jogging was chosen to be the anomalous activity. The reasoning behind choosing jogging as anomalous activity is that it was harder to distinguish from the rest of the activities compared to defining any other activity as anomalous. We observed that the amplitude of the sensor readings obtained from one participant when walking was similar to that of another participant when jogging. We also tested jointly defining several activities as anomalous, but this did not significantly change the outcome.

The timestamp was altered to be the time in between each pair of adjacent point (for each user) in seconds. We noticed that the time difference between some of subsequent points was large, which might be due to the participant taking a break from the current activity and closing the application used to record the sensor data. Since there most likely won’t be any correlation between points sampled at long intervals, the users were split into separate recordings based on if the time difference was larger than ten seconds. The cut-off value (ten seconds) were chosen from histograms of the time differences. We observed that most of the points had a time difference of less than ten seconds, whereas only a few time gaps were larger that ten seconds. The
data was then split into a training- and test set. The training set contains sequences of length 32 corresponding to one of the normal activities from a single user. The test set contains sequences of the same length where half of the sequences correspond to an anomalous activity and half to normal activity. The accelerometer values were standard scaled to zero mean and unit variance based on the mean and variance of the training data. The same standard scaling is then applied to the test data.

3.4 Model Choice

While many different model variants exist that could be suitable for this problem, we are limiting ourselves to some specific models. We are using five different models for experiments. That is a Gaussian mixture model, vanilla LSTM model, LSTM variational autoencoder, LSTM-ODE model and a latent-ODE model. The neural ODE-based models (LSTM-ODE and latent-ODE) are chosen since they have shown great result when modelling irregular and sparsely sampled time series data (Rubanova et al. 2019, Lechner & Hasani 2020, Habiba & Pearlmutter 2020). We use a vanilla LSTM- and an LSTM based variational autoencoder model for comparison to the Neural ODE models since they are common when modelling sequential data and because the Neural ODE models that we used utilise the vanilla LSTM within their architecture. Since the GMM handles each point independently, it does not utilise any context within its modelling. Therefore, the GMM can give an idea how well the LSTM based methods manage to utilise the context of the data.

3.5 Experimental Settings

Three different strategies for anomaly detection were tested.

1. A probabilistic method which does not use context in its predictions (GMM).

2. Two autoregressive methods that classify points as anomalous if their one-step predictions are too far off from the true values (LSTM and ODE-LSTM).

3. Two variational-autoencoders that classify points as anomalous based on the reconstruction errors (LSTM-VAE and latent-ODE).
To understand how well the methods handle irregularly sampled time-series every method is trained on different subsets of the dataset. First off, the full dataset. Then the same dataset sampled uniformly without replacement for different fractions. For 75%, 50% and 25%. This makes it so that the lower the fraction of data is used, the more sparse and irregular it becomes. The timespan of the data is still the same, only less samples over that same period. All the models were implemented using python. The ANN methods were implemented using pytorch (Paszke et al. 2019) and the Neural ODE parts using a library called torchdyn (Poli et al. 2020). Finally, the GMM was implemented using the scikit-learn package (Pedregosa et al. 2011).

### 3.5.1 Probabilistic method

Twenty-seven components and a full covariance matrix were chosen for the GMM based on the Bayesian information criterion (see section 2.2.1). The anomaly score used was the likelihood of each sample under the current fitted model. Since the GMM cannot model correlations between time points, there is no need to feed the data points sequentially at training. Thus, all of the training data are fed simultaneously when training the model.

### 3.5.2 Autoregressive methods

The two autoregressive methods are a vanilla LSTM and an LSTM-ODE. Each of them had a hidden state of size 64, and the LSTM-ODE has a Neural-ODE with a three-layer ANN with \( \tanh \) activation functions as dynamic function. A batch size of 32 was used, and the models were trained for 50 epochs. During training the models are fed 33 consecutive time points and the time difference between them in the case of the LSTM-ODE. The value of the last time point is then predicted from the first 32 and mean squared error is used as a loss function. The residual loss is used as anomaly score (see equation (3.1)).

### 3.5.3 Variational autoencoders

The two variational autoencoders were a vanilla LSTM-autoencoder and a Latent-ODE model. A vanilla LSTM-autoencoder is similar to the Latent-ODE model, but it has a many-to-one LSTM as an encoder and a one-to-many vanilla LSTM as a decoder. All the models had a hidden size of 64. The -ODE has a Neural ODE with a three-layer ANN with \( \tanh \) activation functions as dynamic function. The latent state had size 16. A batch size of 32 was used and the models were trained for 50 epochs. During training, the models were fed
32 consecutive time points and the time difference between them in the case of the latent-ODE. They are encoded into the latent state and then reconstructed back. The ELBO loss is used during training and only the reconstruction loss for the anomaly score.

3.5.4 Settings for the neural ODE based methods

The Neural ODE-based methods have a few more settings. For the LSTM-ODE models a Runge Kutta 4 solver was used. A Dormand prince 5 adaptive step size solver was used for the decoding models with relative tolerance of 0.001 and absolute tolerance of 0.0001. As covered in Rubanova et al. (2019), minibatching these methods is difficult for Neural ODEs. The numerical solvers available cannot solve for different time points for each input sequence in a batch. Instead, the union of all the time points is taken and each input is forward propagated for all the time points in the union. An example is a batch of size two with one sequence to be solved for times (0,1,2,3) and another for times (0,2,4,6). To minibatch, both sequences are solved for times (0,1,2,3,4,6), and then the extra results are discarded.
Chapter 4

Results

This chapter covers the results obtained from the experiments described in section 3.5. The results for all models are presented in the form of ROC-curves and histograms of the anomaly scores. The chapter is divided into three sections, in which five different models are tested. A table is included at the end of the chapter that compares the number of parameters and relative training time for each model.

The human activity recognition dataset was split into two sets, one set with the activity/activities labelled as anomalous and another set without the data labelled as anomalous. 80% of the data from the anomaly free dataset was then randomly selected as training data, and the rest, together with a fraction of the anomalous set, was used as test data. The test data was selected so that half of the data was normal and the other half was anomalous. Dividing the normal and anomalous labelled data equally among the test set was done in order to increase the interpretability of the results. Using balanced data is preferable for ROC-curves since imbalanced data tend to produce similar ROC-curves for different models and might not express the true performance of each model (Saito & Rehmsmeier 2015).

4.1 Probabilistic Method

The result when training and testing a Gaussian mixture model on the dataset can be seen in figure 4.1. The ROC-curve shows similar results for varying fractions of training data. The highest AUC was 0.717, which was obtained when using only 25% uniformly sampled (without resampling) data points for training. For the full data, an AUC of 0.701 was obtained, and the AUC for 75% and 50% of sampled data was 0.69 and 0.691, respectively. The mean of
the anomaly score was the same for all anomalous points, whereas the mean of the anomaly score for the normal points differed a bit which seemed to influence the AUC score.

Figure 4.1 – On the left, we see ROC-curves generated by a Gaussian mixture model for the human activity recognition dataset of varying fractions. To the right, we see the corresponding histograms of the anomaly scores (probability of each point being from the mixture of Gaussian components).

4.2 Autoregressive Methods

Figure 4.2 shows the ROC of a vanilla LSTM model when trained and tested on the HAR data. The best result was obtained for the full training data and gave an AUC-score of 0.886. The scores were lower when trained on the irregularly resampled data. Using 75% and 50% of uniformly sampled data points from the full data (without resampling) for training produced a marginally lower AUC of 0.862 and 0.858, respectively. When sampling 25% of the data, the AUC was lower (0.849). The mean of the anomaly score for both the normal and anomalous points seems to increase the lower the fraction of data. The difference between the means appears to be fairly constant (minimum difference of 0.3 for full data and a maximum difference of 0.41 for 25% of the data). However, by observing the histograms in figure 4.2, the variance increases for the anomaly score for both normal and anomalous points for lower fractions which creates a larger overlap between the classes and thus a lower AUC.
Figure 4.2 – On the left, we see ROC-curves generated by a vanilla LSTM model for the human activity recognition dataset of varying fractions. To the right, we see the corresponding histograms of the residual losses.

Figure 4.3 illustrates the ROC-curves of an LSTM-ODE model on the HAR data. A maximum AUC score of 0.912 was obtained for the full data. Using 75% and 50% of the data produced similar AUCs of 0.898 and 0.892, respectively. The AUC dropped when only using 25% of the data, where the score obtained was 0.866.

Figure 4.3 – On the left, we see ROC-curves generated by an LSTM-ODE model for the human activity recognition dataset of varying fractions. To the right, we see the corresponding histograms of the residual losses.
4.3 Variational Autoencoders

Figure 4.4 shows the ROC-curve generated by an LSTM variational autoencoder model for the human activity recognition dataset with reconstruction loss as the anomaly score. The ROC-curves show that the model trained on the full data produced the highest AUC-score of 0.985. When lowering the amount of data to 75% and 50% by sampling (without replacement), an AUC of 0.983 and 0.977, respectively, was obtained and using 25% of the data gave an AUC of 0.959.

![ROC curves and anomaly scores for different fractions](image)

Figure 4.5 below shows the resulting ROC-curves for the latent-ODE model. The AUC scores for 100%, 75% and 50% of data were marginally higher than for the LSTM variational autoencoder (0.991, 0.988 and 0.982 respectively). For 25% of the data the latent-ODE obtained a similar score as the LSTM-VAE.
Figure 4.5 – On the left, we see ROC-curves generated by a Latent-ODE model for the human activity recognition dataset of varying fractions. To the right, the corresponding histograms of the residual losses.

4.4 Summary

Figure 4.6 summarises the AUC-scores for the different models when varying the irregularity and sparseness of the data. The solid, dashed and dotted lines correspond to the variational autoencoder-, autoregressive- and statistical models, respectively. The red lines correspond to the Neural ODE based models. The highest AUC-score was obtained for the variational autoencoder models in which the latent-ODE got a slightly higher score than the LSTM-VAE. For the autoregressive models, the LSTM-ODE model got a higher AUC-score than the LSTM model. The probabilistic Gaussian mixture model got the worst score for all fraction by a large margin.
Figure 4.6 – Summary of the AUC-scores obtained for the five different models for various fractions of the human activity recognition data. A fraction of one corresponds to the full dataset and a fraction of x corresponds to uniformly sampling x*100% of the full data without replacement.

The figure shows that the variational autoencoder models (solid lines) were more consistent for sparser and more irregular data. The LSTM-ODE seems to be a bit less sensitive to sparsity and irregularity when compared to the vanilla LSTM model (dashed lines). The Gaussian mixture model obtained the highest AUC-score when only using 25% of the data, indicating that it is not as affected by the irregularity and sparsity of the data.

Table 4.1 compares the five models based on the number of parameters and training time relative to the LSTM model. The metrics compared in the table are obtained for the human activity recognition dataset when using the experimental settings explained in section 3.5. The relative training time was obtained by measuring the average wall clock time of an iteration (one batch update) and then dividing by the average time of an iteration for the vanilla LSTM model.
<table>
<thead>
<tr>
<th>Model</th>
<th>Parameters</th>
<th>Training time (relative to LSTM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMM</td>
<td>269</td>
<td>-</td>
</tr>
<tr>
<td>LSTM</td>
<td>17.900</td>
<td>1.0</td>
</tr>
<tr>
<td>LSTM-ODE</td>
<td>34.600</td>
<td>12x-48x</td>
</tr>
<tr>
<td>LSTM-VAE</td>
<td>40.500</td>
<td>4x</td>
</tr>
<tr>
<td>Latent-ODE</td>
<td>46.900</td>
<td>24x-84x</td>
</tr>
</tbody>
</table>

Table 4.1 – Comparison of the different models for HAR data based on the number of parameters and training time relative to the LSTM model

The relative training times for the neural ODE based methods were significantly higher than the other method (ranging from 12 to 84 times longer than a vanilla LSTM model). The range of relative training time for the Neural ODE based models is because it was slower to train these models for lower fractions of data. This is because the sparser the data, the longer the time interval between points and the more variation there is of time points to solve for each batch. Thus, it requires more computations to numerically integrate, and more calculations are required due to the mini batching method.
Chapter 5

Discussion

The result obtained by the Gaussian mixture models indicates the model is not influenced by the sparsity or irregularity of the data. This is likely because the GMM does not model any temporal dependency/correlation of the data and does not require as much data relative to a neural network model to train. Although it is robust to irregularity and sparsity, it is worse at detecting anomalies than the other models. As shown in the histograms in figure 4.1, the model did not create a good spread of the anomaly scores for the normal and abnormal points. The considerable overlap of the anomaly score makes it impossible to find a suitable threshold value without many false negatives or false positives.

The results show that the LSTM-ODE performed better than the vanilla LSTM model. The ROC curves show improved AUC scores compared to the vanilla LSTM model in figure 4.2. The performance did not seem to deteriorate as fast when increasing the irregularity and sparsity of the data compared to the vanilla LSTM as can be seen from the AUC when utilising 75% and 50% uniformly sampled data points. The relative change of the AUC compared to the full data was -1.5% for 75% of the data and -2.2% for 50% of the data. Comparing this to the vanilla LSTM, we got a relative change of AUC of -2.7% for 75% of the data and -3.2% for 50% compared to the full data. The histograms for the vanilla LSTM and LSTM-ODE appear to share the same characteristics. The means and variances of the anomaly score increase for both normal and anomalous data when decreasing the fraction.

The result for the variational autoencoder models (LSTM-VAE and latent-ODE) showed that they obtained higher AUC-scores than the autoregressive models. Both models created a good spread of the anomaly scores between the normal and anomalous data, as can be seen in the histograms in figures
The anomaly score (reconstruction loss) for the anomalous points had a very high variance. The variance for the normal points was initially quite low but increased for lower fractions of data. Both models seem to be robust to sparser and more irregular data (see figure 4.6). The AUC score for 100%, 75% and 50% of data was higher for the latent-ODE than the LSTM-VAE, and for 25%, both models obtained a similar score. The drop in performance when using 25% of data indicates that it is more challenging to model the context within the data for this low fraction. Since the LSTM variational autoencoder model produced such a high AUC-score, it leaves a small room for improvement. This makes it harder to see the benefit of using the Latent-ODE. Therefore, it might be appropriate to use another dataset in which the normal and anomalous points are harder to distinguish.

An interesting pattern we found was the relationship between the distributions of the anomaly scores and the AUC-scores. A limiting factor in the performance of all the methods was how as the normal anomaly scores got lower, the anomalous scores also decreased. The main contributor to the increase in AUC score was the decreased variance of the distributions for higher fractions, which allowed for better separation. As the means of the anomaly scores for the normal data doubled or even tripled, the drop in the AUC-score dropped by only a few percentage points for lower fractions. This relationship shows how important it is, not only for the anomaly detector to be good at predicting/reconstructing the normal data, but it also has to be bad at the same task for the anomalous data. A perfect reconstructor of the normal data is useless at anomaly detection if it is also perfect at reconstructing the anomalous data.

The time it took to train the different models varied based on the defined settings. However, utilising a Neural ODE variant seems to affect the time spent training substantially. This is because ordinary differential equations are computationally expensive to numerically integrate. We noticed that using an adaptive step size solver seemed to gradually slow down the training as the epochs progressed. This is because the ODE-solver attempts to solve the ODE within a given tolerance bound. As the training progresses, the dynamic becomes more complex which forces the solver to take smaller steps which increases the training time. We tried both the adjoint method and direct backpropagation to train the networks. However, using the adjoint method was slower and created a higher loss than directly backpropagating through the internal operations of the ODE-solver. This is because using the adjoint method for updating the parameters requires integrating two differential equations, one for the forward pass, and one for the backwards
pass. This introduces extra numerical error and computation time. Since the utilised memory during training is not a problem for the batch size we used, we chose to backpropagate directly instead of using the adjoint method.

There is potential to speed up the Neural ODE based methods. The method is new so the current implementations available are not fully optimised. The LSTM module within pytorch is heavily optimised and very efficient. The modifications necessary to go from a vanilla LSTM to an LSTM-ODE require the use of the LSTMCell module, a less optimised building block of the LSTM module. A fully optimised implementation of the LSTMCell and Neural ODE combination could speed up the calculations. The mini batching method of the Neural ODEs was also a bottleneck as the extra calculations slowed down the training, especially for lower fractions. An adaptive ODE solver which can solve for different time points within a batch simultaneously would provide a large speed boost.
Chapter 6
Conclusions

This thesis investigated the feasibility of using Neural ODE based methods for anomaly detection on the edge. The data investigated was multivariate time-series with collective anomalies that corresponded to abnormal behaviour. We introduced two different Neural ODE based methods, the ODE-LSTM and the latent-ODE. Their performance was then compared to two similar methods based solely on vanilla LSTMs, an LSTM autoregressive model and an LSTM variational-autoencoder, and a gaussian mixture model that provided a comparison to a predictive method that does not utilise context.

We introduced metrics for evaluation and one dataset for training for the comparisons. The dataset was the WSDM human activity recognition dataset which we used for semisupervised experiments. One activity was chosen as anomalous and the others as normal. We trained the methods to predict or reconstruct sequences from normal activities. For anomaly detection, we fed the methods either anomalous or normal sequences with the hope that they would perform worse at their task on the anomalous data.

The ODE-based methods proved to be slower and more memory consuming than the other methods. That is a drawback for edge devices which have limited computational power. Their accuracy was slightly better than their LSTM counterparts for the data used in this report and they were more robust to irregular data. However, the differences were small.

Neural ordinary differential equations are an interesting approach to neural networks. They have been utilised with good results in a variety of tasks. However, from the results presented here on the HAR data, we found that the increased accuracy and robustness to irregular data, came with the cost of large increase in computational effort.
6.1 Future Work

There is much potential for interesting further experiments. We list here some approaches that could be interesting to investigate further.

Further improving the methods

As mentioned in the delimitations (chapter 1.4), more complex variations of Neural ODEs have been developed. Implementing them and comparing their performance to the vanilla Neural ODE could give an interesting insight into the methods. Examples include Neural Jump Stochastic Differential Equations (Jia & Benson 2020) and Neural Controlled Differential Equations (Kidger et al. 2020).

Trying different datasets

Anomaly detection is very problem dependent. The different types of data, types of anomalies and types of training possible on said data makes it so that the strategy for the anomaly detection needs to be evaluated on a case by case basis. Doing similar experiments on different datasets would provide more insight into the performance of the methods in other settings.

While the HAR dataset was standard scaled and smoothed, further improvements could be made by pre-processing the data more or doing feature extractions. Some ideas include calculating the derivatives or integrals of the data over periods of time and dimensionality reduction, e.g. using principal component analysis. Due to time limitations, that was not done in this thesis but would be interesting to see if better results could be obtained.

Trying different anomaly scores

This thesis has worked with simple anomaly scores such as residual loss and reconstruction error. These metrics reduce the score down to a single number from a multidimensional error. Separating the classes in a single dimension can be difficult as seen in the anomaly score histograms. Having more dimensions to find a separating hyperplane could be helpful. For example, having a separate anomaly score for each feature or combinations of features. Zong et al. (2018) introduced a way of adding a GMM after an autoencoder to utilise the distributions of the errors for anomaly detection. Such experiments with the latent-ODE would be interesting.
52 | Conclusions
References

Cao, K., Liu, Y., Meng, G. & Sun, Q. (2020), ‘An overview on edge computing research’.


Mattern, F. & Floerkemeier, C. (2010), From the internet of computers to the internet of things, Vol. 6462 LNCS.


Nielsen, M. A. (2015), Neural Networks and Deep Learning, Determination Press.
URL: http://neuralnetworksanddeeplearning.com


Appendix A

SKAB data

A.1 The data

The Skoltech anomaly benchmark (SKAB) (Katser & Kozitsin 2020) is a dataset specifically designed for benchmarking anomaly detection algorithms. The data was gathered using a testbed of a water circulation system consisting of water pipes with adjustable valves, a tank with water, a water pump, an electric motor and various sensors. Faults could be injected into the system by adjusting the various part of the testbed. Examples include increasing the temperature of the water in the system or partly closing the valves.

The data consists of 35 files containing multivariate time-series describing the status of the testbed with one point per second. The first one is an anomaly-free file where the system runs for a little less than three hours under normal conditions and can be used if one wants to perform semi-supervised training. The next 34 files each contain data sampled during a time window of roughly 20 minutes. All of these files include anomalies. Each file starts at a normal state with no fault. After some time a fault is injected, and the system runs in this faulty state for a while until the fault is removed and the system again runs normally. For unsupervised training, the first 400 time points of each file can be used for training and to simulate real-world implications of training in an unsupervised manner, some files contain errors within the first 400 seconds. The test data is then all time points after the 400-second mark in each file. There were some failed samples, as can happen with real-world sensor data, meaning that there were some irregularities in the timesteps between samples. However, 95% of the timesteps were 1 second, so the dataset is mostly regularly sampled. Each data point contains information about eleven different attributes. The attributes, together with their corresponding data type,
unit and outcome range, can be seen in table A.1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datetime</td>
<td>Date and time of the measurement</td>
<td>-</td>
</tr>
<tr>
<td>Accelerometer1RMS</td>
<td>Vibration acceleration</td>
<td>g units</td>
</tr>
<tr>
<td>Accelerometer2RMS</td>
<td>Vibration acceleration</td>
<td>g units</td>
</tr>
<tr>
<td>Current</td>
<td>Amperage on electric motor</td>
<td>A</td>
</tr>
<tr>
<td>Pressure</td>
<td>Pressure in the loop after the water pump</td>
<td>Bar</td>
</tr>
<tr>
<td>Temperature</td>
<td>Temperature of the engine</td>
<td>°C</td>
</tr>
<tr>
<td>Thermocouple</td>
<td>Temperature of the fluid</td>
<td>°C</td>
</tr>
<tr>
<td>Voltage</td>
<td>Voltage on electric motor</td>
<td>V</td>
</tr>
<tr>
<td>RateRMS</td>
<td>Circulation flow rate of the fluid</td>
<td>L/m</td>
</tr>
<tr>
<td>Anomaly</td>
<td>Indicator whether point is anomalous</td>
<td>-</td>
</tr>
<tr>
<td>Changepoint</td>
<td>Indicator whether point is a changepoint</td>
<td>-</td>
</tr>
</tbody>
</table>

Table A.1 – Features of the SKAB dataset.

The anomalies within each file are collective anomalies as they correspond to a prolonged run of faulty behaviour. Figure A.1 shows an example of the behaviour of a few features.

Figure A.1 – An example of an anomaly in the SKAB data. Three features are plotted from a single experiment. The period corresponding to an anomaly is shaded red.

The file plotted here corresponds to an experiment where higher temperature water is added into the system. Both the thermocouple and flow...
rate features are affected by the water input but with a delay as it takes time for the hotter water to reach the thermometer and flow rate sensor within the system. The thermometer feature shows the ongoing long and short term patterns within the other features that do not have an apparent change in behaviour for this specific anomaly. From this example, it is clear that detecting anomalies in this dataset is hard as it would be hard for a human to estimate the start and end of the anomalous period from the figure.

The training dataset was obtained by concatenating the first 400 seconds of each file that had anomalies. Similarly, the test set is obtained by concatenating the subsequent points after the 400-second mark from each file. Each feature is standard scaled to zero mean and unit variance from the mean and variance of the training data. The changepoint column was also removed. As the data was very noisy, a 10 second rolling average was applied. We felt that the number of anomalies in the training data was not enough to warrant calling it unsupervised learning. The only file that included anomalies in the first 400 seconds was the first file, so instead we removed that file and used semi-supervised learning. An added benefit was that the number of anomalous and normal points after the 400-second mark was approximately equal in the dataset, which allowed for accurate comparisons using the ROC curves.

A.2 Results

This section covers the results obtained when training and testing the models on the Skoltech anomaly benchmark dataset. Training was done on the first 400 seconds of each file and testing was done on the rest. The model settings were the same as for the HAR data, except the hidden state size, and latent size were twice as large (128), while the input sequence length was half that of the HAR data (16 seconds). The most suitable number of components, and covariance type for the GMM was the same as for the HAR data. The Mahalanobis distance was used for the anomaly score.

In the results plot, the median is marked instead of the mean on the histograms. The distributions of the anomaly scores are very right-skewed, and the medians provide clearer visual comparisons.

A.2.1 Probabilistic Method

Figure A.2 shows the performance of the GMM on the SKAB dataset. A sigmoid function was applied to the anomaly score for easier visualisation.
Figure A.2 – On the left, we see ROC-curves generated by a GMM model for the SKAB dataset of varying fractions. To the right, the corresponding histograms of the residual losses.

Similar to the HAR data the GMM is robust to the sparsity and irregularity. The AUC is around 0.75 for all fractions.

### A.2.2 Autoregressive methods

Figures A.3 and A.4 compare the results of the autoregressive methods on the SKAB data.

Figure A.3 – On the left, we see ROC-curves generated by an LSTM for the SKAB dataset of varying fractions. To the right, the corresponding histograms of the residual losses.
Figure A.4 – On the left, we see ROC-curves generated by a LSTM-ODE for the SKAB dataset of varying fractions. To the right, the corresponding histograms of the residual losses.

While the anomaly scores are much lower for the ODE-based methods, the AUC scores are not better. The better performance of the ODE based methods transfers over to the anomalous class, and the separation between the two classes does not improve. An interesting phenomenon can be seen for the lowest fraction where the anomaly scores are much worse than for the full dataset. Still, the relative increase is higher for the anomalous class, which lead to an improvement in the AUC score.

**A.2.3 Variational autoencoders**

Figures A.5 and A.6 compare the results of the variational autoencoders on the SKAB data.
Figure A.5 – On the left, we see ROC-curves generated by a LSTM variational autoencoder for the SKAB dataset of varying fractions. To the right, the corresponding histograms of the residual losses.

Figure A.6 – On the left, we see ROC-curves generated by a Latent-ODE for the SKAB dataset of varying fractions. To the right, the corresponding histograms of the residual losses.

The LSTM-VAE performs better for the lower fractions. The histogram of the anomaly scores shows what happens under the hood. The better lower the anomaly score gets for the normal sequences, the lower it gets for the anomalous class. Therefore, no clear threshold appears between the classes. For the lowest fraction, the anomaly score for the normal class is higher but the score for the anomalous is much higher which negates the negative effects. The Latent-ODE seems to perform better and be more robust to the decreasing fractions. However, the benefit does not show as it gets better for both the anomalous class and the normal one.
A.3 Discussion

A disclaimer here is that the GMM outperforms the more complex methods. This indicates that the dataset is not well suited for the LSTM methods. Therefore, the comparisons made are not necessarily generalisable to datasets where the LSTM’s are well suited to the problem.

The anomaly score histograms do add valuable information. Similar to the HAR data, the benefit of a lower anomaly score for the normal class does not translate to a better anomaly detector if the score for the anomalous class also decreases. A clear example is figure A.6, where the AUC increases as the median of the anomaly score increases. What seems like increasingly bad variational autoencoders perform better at anomaly detection as they get even worse at reconstructing the anomalous class relative to the normal class.

This could be because the difference between the anomalous class and the normal class are small. In figure A.1, one can see that the anomaly stops as soon as the anomalous change is removed from the system. The flow rate in the figure starts decreasing but is still in what could be considered an abnormal state as it decreases down to the original value. Similarly, there seems to be a delay before the anomaly is detected in the data. This data labelling means abnormal and normal sequences can be very similar. This similarity means that good LSTM-based methods can learn anomalous sequences when trained on normal sequences.

While the results could not be included in the main part of the report, we do belief they help with understanding of the inner workings of these anomaly detection models.