EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Sequence Prediction in Real-time Systems

Master Thesis Project

Cody Arets

Supervisor:
- Dr. Mitra Nasri

V7.0

Eindhoven, September 2022

Abstract

Real-time and embedded systems are becoming increasingly larger in terms of software components and more complex in terms of design. This brings a growing need for tools and techniques for monitoring the runtime behaviour of the system and its internal activities. Typically, the main goal of these tools and techniques is to detect anomalies that can jeopardize the safety and security of a real-time embedded system.

In particular, in this thesis, we are interested in developing runtime monitoring tools that are able to predict future events in a real-time system. Here are some of the relevant questions in the context of our work: (i) what is the next message (or task) that will appear on the network (or occupies a certain resource)? (ii) When does a certain message reappear on the network (or access a certain resource) again? Answering these questions allows us to detect whether the expected behaviour of the system still matches the specification of the system and whether there is an anomaly in the system's performance.

The aforementioned questions are a part of a bigger problem, i.e., event prediction in time series. An accurate prediction of the next event (symbol) that will appear on a time series is a complex problem, particularly when there are a lot of uncertainties in the behaviour of the underlying system/platform/applications. In reality, the observations we may have from a system are also noisy and might be incomplete. This means that a practical solution must not only be *accurate* but also *robust* against incomplete input data and noise.

In this work, we will apply machine learning and, in particular, neural networks, to predict future messages that are going to be transmitted on a *Controller Area Network* (CAN) in a distributed real-time system. Our main research questions are as follows:

1. What type of neural networks may be a better fit for our problem?
2. Should we do the feature extraction by ourselves (using expert knowledge) or should we let the machine learning solution finds that for us? In the former case, a follow-up question is what features should we use? and in the latter case, the follow-up question is how to apply the neural network technique to a problem with a variable input size?
3. To what extent is it possible to make an efficient solution for event prediction?

This report aims to give background information, a literature review, a problem definition, and some initial steps and ideas for our solution approach.

Contents

Chapter 1

Introduction

1.1 Area/General Idea

John C. Knight[1] defines safety-critical systems as systems whose failure could result in loss of life, significant property damage, or damage to the environment. The development of more safety-critical systems and the ever-increasing presence of real-time systems in the embedded systems industry highlights that correct operation, high safety, and high security are natural requirements.

Because these systems are often the target of attack [2]. It brings a growing need for tools and techniques for monitoring the runtime behaviour of the system and its internal activities. One of the main goal of these tools and techniques is to detect anomalies that can jeopardize the safety and security of a real-time embedded system and to respond accordingly. In particular, in this thesis, we are interested in developing runtime monitoring tools that are able to predict future events in a real-time system.

Predicting future events is especially useful for embedded systems which are distributed. For example, in automotive systems, devices called *Electric control units*(ECU) are connected over a network. The ECUs communicate over this network by broadcasting messages to it. From a safety and security perspective, knowing what messages are expected to come next is valuable information in checking the correct functioning of the system, detecting anomalies, and monitoring safety properties.

The *Controller Area Network* (CAN) bus is an example of such a network. The CAN bus employs the CAN bus protocol in which messages are identified by a unique ID. For the CAN bus, the event prediction problem becomes, given as input a time series of previous message ID values, and possibly other data, predict the message ID of the next message that will appear on the CAN bus.

1.2 Problem description

In this work we assume a single bus on which messages with a message ID appear. The goal is to be able to accurately predict which message ID will appear next. We do this by observing the bus and generating a trace of messages. The traces have a certain length w, also referred to as the window length, for the number of messages that are recorded sequentially and are used as input. The input feature is an ordered list of elements in which elements can repeat, consisting of a trace of message ID's taken from a set $x \in \{ID_1, ID_2..., ID_n\}$ and the other input feature is a list of the time in micro seconds between the start of each message. As the output we aim to predict one message ID, $y \in \{ID_1, ID_2..., ID_n\}$ which is the most likely to appear after the recorded trace.

As an example, figure 1.1 depicts a trace with window length $w = 5$. The first input feature is the IDs in order $id\_feature = [1, 3, 2, 3, 4]$ and the time feature is $time\_feature = [3.0, 3.0, 2.0, 2.0]$. The desired output is $y = 3$.

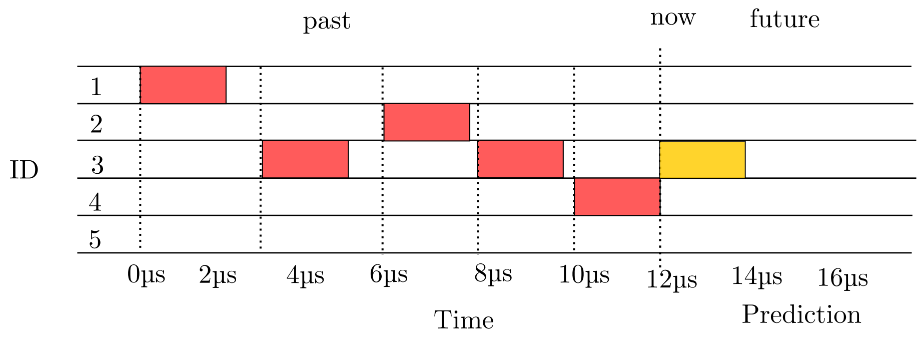


Figure 1.1: Depiction of a CAN trace prediction, a trace is displayed where several messages with a message ID are present, the yellow message is a prediction of a model.

1.3 Motivation

To solve the problem of sequence prediction in real-time systems we investigate related fields. A summary of our findings is presented in this section, which will not only describe the solution idea but also motivate it.

In any real-time system, tasks or devices can have many different properties and states. So even though there is often periodicity to them, it is not trivial to determine which one comes next or when a message appears on a network. Vădineanu and Nasri [3] have used regression-based machine learning to infer some of these properties, in particular the periodicity of messages (or tasks). It is likely that machine learning can be used to infer more of these properties or do it more accurately.

Both time series forecasting and sequence prediction have advanced greatly in recent years due to the increased computational power available for machine learning. In 2018, Makridakis et al. [4] found in their comparison that machine learning models are still not as accurate as statistical methods on 1045 monthly time series. Independently Parmezan et al. [5] found in their 2019 evaluation that machine learning offers similar or better results than state-of-the-art statistical methods for time series forecasting. Furthermore, the field of sequence modeling has been revolutionized by machine learning methods. In particular, *Recurrent Neural Networks* (RNN) have brought improvements in many fields such as translation [6], speech recognition [7], and biological sequence analysis [8]. Our hypothesis is that machine learning is suitable to our problem and can outperform statistical methods.

There are several challenges in the way of applying machine learning to the problem of sequence prediction of real-time systems: (i) Pascanu et al. [9] argue why RNN are hard to train properly in their paper on the difficulty of training recurrent neural networks. (ii) Because of the number of different machine learning models, it is a challenge to find the most suitable model. (iii) Another challenge is that the computational complexity of a solution would have to be small if it is to be used in any real-time application. Makridakis et al. [4] argue that statistical methods in prediction of monthly time series often outperform machine learning methods with significantly less computational cost. (iv) Lastly, a challenge is often to collect enough quality data when using machine learning. But data is widely available by monitoring real-time system busses or logs.

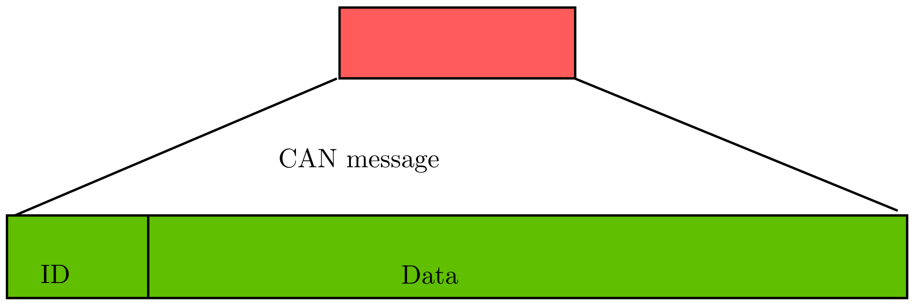


Figure 1.2: Depiction of a CAN message that consists of a message ID part and a data part.

1.4 Gap in state of the art

In chapter 3, we will discuss in depth the related works which exists. This section aims categorize our related work and briefly explain how they are similar and yet different from our work. First, *intrusion detection systems* (IDS) are presented because of the very active state of research of this field and because these systems often rely on predictions. Then works on predictions of the task period are discussed. Following that is a discussion the works on forecasting on time series and how it is related to this work. Finally, the work on sequence modelling is presented.

Intrusion detection systems, the complexity of in-vehicle networks is increasing. This causes a security risk as these systems lack the proper tools against cyberattacks. *intrusion detection systems* (IDS) exist to monitor a network and notice malicious actors in order to take measures for protection of the network. In recent years research on these systems has increased, Wu et al. [10] wrote in their survey about 20 methods for IDS. They differentiate various strategies which are discussed more in chapter 3. Most related to this work are the methods that applied machine learning. They show that machine learning can be successfully applied to the types of data sets we are investigating. Each work uses a different model and most of them apply the input of the intrusion detection problem in a different way. A particular notable model is the *long short-term memory*(LSTM), applied by Taylor et al. [11]. For each CAN message ID they train a LSTM model to predict the chance of an intrusion. Another use of LSTM is by Pawelec et al. [12]. They aim to predict the contents of an entire message to help with intrusion detection. Both works use the entire message bits as input but with a different number of messages. They differ from this work as they only focus on predicting correct behaviour and comparing it to the actual behaviour of a system, not the more general problem of predicting what message ID comes next on a network.

Task period, Works exist to extract the task period from execution sequences [13, 3, 14]. The task period is one of the most vital properties to know when the task will reappear in a sequence. In reality, however, systems have many uncertainties in their timing behaviour. These are caused by, for example, sporadic tasks, deadline misses, release jitters and in the case of fixed-priority scheduling interruptions from higher priority messages. Even if someone can predict periods, they might not be a good indication of when a message appears on a bus.

Forecasting time series is using a model to predict future values based on previously observed values. It often concerns real world prediction problems like, weather, financing or traffic. Problems in these categories rely on real-valued continuous data and are not directly comparable to the sequence problem in real-time systems. However, the field is well researched and contains many surveys. One of the surveys comes from the Makridakis competitions [15, 4]. It is a research competition where different models compete for the best forecasting results, providing valuable information on the effectiveness of these models. Furthermore, there are two surveys that focus on deep learning [16, 17]. These deep learning approaches are especially interesting because the models are more versatile than statistical approaches. In the field of time series forecasting there are more general surveys [5], which compare statistical and machine learning methods, and surveys focusing on certain methods like Fuzzy time series [18] and Support vector machines [19]

Unfortunately, the works mentioned above do not analyse or use sequences. The predicted values are not discrete classes. This is not in line with CAN messages which are discrete classes. Using pre-processing on CAN data it is possible to convert a data sequence to a relevant time series problem and therefore apply the techniques described. However, when converting data like this information will be lost, this means using this method we might not get good results.

Sequence modelling, Neural networks are employed for sequence modelling and making predictions in other areas sensor prediction [20], trace restoration [21, 22], and emergency event prediction [23]. We found no work which aims to predict messages on a network directly. One of the most encountered solutions is to use *long short-term memory*(LSTM) networks. Several improvements for training LSTM have been suggested [24] and architecture search [25] has been applied to find the best candidates for models in arithmetic problems, XML modelling and a word level language modelling task. Others used *temporal convolution networks*(TCN) for critical care prediction [26] or showed that TCN outperform *Recurrent neural networks*(RNN) [27]. RNN themselves are widely reviewed [28, 29, 30].

The findings of these papers all show the potential of machine learning and especially the mentioned models in the area of sequence modelling.

1.5 Research Question

The problem description (see Sec. 1.2) results in three main research questions in this work, they are as follows:

1. **What type of neural networks may be a better fit for our problem?** To answer what is the best fit for our problem we aim to setup experiments with at least three machine learning algorithms identified in preliminary research. We found similar prediction problems in different field which are solved using different algorithms, like those in the papers of Makridakis et al. [4] and Parmezan et al. [5]. They discussed both machine learning and statistical methods. Further elaboration is found in chapter. 2. We identified three classes of neural networks that are of particular interest from the machine learning field. LSTM, GRU, and TCN. They are further discussed in chapter 4 Sec. 4.5.

2. **Should we do the feature extraction by our-selves (using expert knowledge) or should we let the machine learning solution finds that for us? In the former case, a follow up question is what features should we use? and in the latter case, the follow-up question is how to apply the neural network technique on a problem with a variable input size?** Most machine learning algorithms rely heavily on how the input is structured. Features are independent variables which are used as input. These features will be constructed from the input data. Two possibilities are clear so far. Possibility one is to convert data into the right format by a method called one-hot encoding. Possibility two is to let the model train to find the best vector representation it can find. However, this adds complexity to the learning phase of algorithms and can reduce efficiency of learning. An alternative is to extract the features manually (e.g., based on an expert's knowledge). The implications of this choice will be discussed in detail in chapter 4.

3. **To what extent is it possible to make a solution perform efficiently?** One of the goals of the project is to construct a solution which does not only produce effective results but also to have a program with a low run time. It is likely any practical application of a predictive model will be constrained by embedded hardware as it would run on the same hardware the CAN bus protocol runs. We classify our solution as efficient if the time to predict a message is less than the time it takes a new message on average to arrive on a network. Around $0.5ms$ on average depending on the data set that is used.

1.6 Organization of report

After this extensive introduction, this report will provide a background section (see Chapter 2) into all of the related subjects. Time series will be explained as well as several machine learning models. In particular neural networks are focused on. Furthermore, the CAN protocol is investigated. All related work will be discussed by giving summaries of all important papers with related subjects or ideas(see Chapter 3). The solution idea is discussed (see Chapter 4. Furthermore, the results of the experiments are discussed to explain how well our solution works, what are its advantages and its flaws. (see Chapter 5). Finally, a conclusion is given with thoughts on future work (see Chapter 6).

Chapter 2

Preliminaries

In this section, all necessary concepts are explained in detail. First, time series will be discussed and how they have been predicted in the past (see Sec. 2.1). Then an explanation of neural networks will follow. In particular recurrent neural networks and *long-short-term-memory* (LSTM) networks will be discussed (see Sec. 2.2). Finally, there is a detailed overview of the CAN protocol (see Sec. 2.4).

2.1 time series

A time series is a sequence of data taken from equally spaced points in time [16]. Often made by taking continuous phenomena and transforming them into discrete data. The most common and researched time series are continuous and consist of real values. As many things can be modelled as a time series, for example, temperature, stock prices and events of a sensor being able to predict the next value of values in a time series is immensely useful in many applications. Classical methods try to decompose a time series into its component for predictions [16].

Torres et al. describe these three components in detail [16]. Trend, seasonality and irregular. A trend is the general movement that the time series exhibits during the observation period, without considering seasonality and irregularities. Seasonality is the variations that occur at specific intervals and residual or irregular components are all values that remain after removing the previous two components. Time series can also be either univariate (only one value per unit of time) or multivariate (multiple values per unit of time).

Autoregressive Integrated Moving Average (ARIMA) are a well-established class of statistical time series forecasting models. ARIMA models are used in many applications like reliability forecasting [31], supply chain modelling [32] and stock price prediction [33]. ARIMA combines several observations about time series to model correlations. First, autoregressive models assume that each value of a time series depends on the weighted sum of the product of previous values. Moving-Average models the importance of the mean. However, ARIMA is poor at predicting series with turning points. And not always suitable for non-linear series [5].

Fortunately, another technique does not suffer the same disadvantages, *Artificial neural networks* (ANN) are not limited by linearity and are able to contest statistical models [34]. They do not make any assumptions on the distribution of the data [5] and can deal with time series in a scalable way [16]. However, unlike statistical methods, ANN's as a supervised learning method need a lot of data to be trained. It is hard to reason about the internal workings of a neural network after they have been trained and data has to be represented in a way that the network can work with it. What follows is a more detailed discussion on neural networks.

2.2 Neural networks models

Neural networks (NN) are computational learning models. They are built up out of a network of nodes. In the simplest form these nodes are perceptrons (equation (2.1)). Each connection contains an associated weight (w_i) to reflect how important an input is. The data inputs (x_i) are summed and multiplied by their weight. There are n number of inputs and weights. After this, a bias (b) can be added. The output (y) is then computed, usually a non-linear activation function is applied when the output is used as an input for another perceptron.

$$y = \sum_{i=1}^{n} w_i x_i + b \tag{2.1}$$

By chaining multiple perceptrons together in so-called hidden layers a feed-forward network is created. The hidden layers do not directly interact with the input or output. Deep neural networks can consist of many such layers.

The input for a neural network model is often called a tensor. A tensor is a n-dimensional array which stores the input information.

By supervised learning, neural networks can learn from input and output data by adjusting their parameters to approximate arbitrary continuous functions [29]. This is given by the universal approximation theorem. Unfortunately, it also states that such a network can grow exponentially in size depending on the function. This means that we are limited and cannot approximate any function practically.

Training a neural network is done with backpropagation. A loss function needs to be defined see Sec. 2.2.2. This function can calculate the error rate between test data and an output from the network. The backpropagation algorithm then calculates the gradient of the weights with respect to the loss function. This way the weights can be adjusted to perform better. When training overfitting can occur. This happens when the model is too highly optimized on the training data and loses its ability to generalize on new data. Underfitting is the reverse problem if not enough data is used in training to generalize. The training process can be tuned by many hyperparameters. A big challenge in using neural networks is finding or choosing the correct hyperparameters.

2.2.1 Cross-validation for time series

In machine learning, cross-validation is used to estimate how well a model will perform on actual data by sampling a small part of the data. This removes bias from the training procedure. Most commonly k-fold cross-validation is applied. In k-fold cross-validation, the parameter k determines in how many groups the training data is split. The groups are randomly shuffled and one is taken out as the validation set. The remaining groups are used to train the model. After this, the randomly chosen group is used to evaluate the performance of the model. Each group is taken out once and given a chance to serve as validation data. The results are averaged to get a final validation result.

This method is not suitable for time series. Two problems present themselves. First, when splitting the data into several smaller parts set the temporal continuity at the cut is no longer correct. Secondly, it is not possible to randomly shuffle the parts as the order matters.

A special method of cross-validation exists for time series. The data is again split into k parts. The first split serves as the first training data set and the second part is used for validation. This process continues, taking more training data into account until all parts are used i.e. the second training procedure uses the first two parts as training and the third as validation. Only the first part is only used for training and not validation. Using time series cross-validation the validation set is always in chronological order compared to the training and there are no cuts in the data set.

2.2.2 Loss function

For training a neural network a loss function is required. This function measures how well a tensor x corresponds to a target tensor y. It is used to evaluate how well a machine learning algorithm performed by using the output of a model as a tensor x. There exist many different methods to calculate the loss. L1, L2 and cross-entropy loss are discussed next. The L1 loss norm measures the mean of the absolute error between all elements. The error is calculated by subtracting i.e. $x_i - y_i$. The L2 loss norm however uses the mean of the squared error. This punishes outliers because the error gets squared higher error values produce worse losses. Another criterion is the cross entropy loss which is often used in classification models. For each class in a classification problem, a probability is produced by the network. To compare this probability to the actual desired output a logarithmic scale is used to yield a higher error for incorrect probabilities close to 1. The complete calculation goes as follows.

$$Loss = -\sum_{i=1}^{n} y_i * log(x_i)$$

where n is the number of classes.

2.2.3 Recurrent Neural Networks

Recurrent Neural networks (RNN) are a type of neural network specifically designed for sequence modelling [35]. They adjust the traditional model by feeding the output back into the input step [30]. This leads to RNNs having an advantage in certain situations over standard feed-forward networks. In particular their ability to retain information from a context window of arbitrary length [28]. Retaining information makes them suitable for problems with a temporal dependency [16]. Furthermore, can also handle variable length sequence input [36].

The input of a recurrent network is a sequence described as $x = (x_1, x_2, ..., x_n)$. The hidden state $h = (h_1, h_2, ..., h_m)$ of a network is iteratively computed.

$$h_t = f(W_{xt} * x_t + W_{ht} * h_{t-1}) \tag{2.2}$$

where W is a weight matrix, f is usually a sigmoid or hyperbolic tangent function. The output of the network $y = (y_1, y_2..., y_n)$ can be computed from the hidden state.

$$y_t = W_{yt} * h_t \tag{2.3}$$

Biases can be added after multiplying a weight with an input or hidden state.

RNNs have to deal with the problem of vanishing and exploding gradients [37, 38]. This problem occurs when training a network. Because errors can compound quickly as every output is reused as an input gradient values in a network can quickly vanish and have no notable effect on the long-term output. In the opposite scenario, a value can become exceedingly large. This prevents the network from learning properly and might not produce the desired output [36].

2.2.4 Long Short-Term Memory

Different models aim to deal with the vanishing and exploding gradient problem. The most notable ones are *long short-term memory* (LSTM) [38] and Gated Recurrent Unit(GRU) [5]. It is known that these models do require great amounts of data and their parameterization is complex and expensive [5].

An LSTM unit is built up out of an input gate i, an output gate o and a forget gate f. The variables j and c compose the memory cell. And lastly, h is the hidden state. The hidden state keeps track of the current state and the memory of the cell. The forget gate decides what information in this memory gets removed from the state in a unit. In much the same way the input gate combines new input with the hidden state to decide what information should go into the network. Finally, the output gate decides what the next hidden state will be.

Many variations on LSTM exist. The most important variation is called the peephole LSTM. Where extra connections let the gates not only depend on the hidden state but also on the memory cell c. The following formulas define a common network without peephole connections:

$$\begin{aligned} i_t &= \phi(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\ o_t &= \phi(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\ j_t &= \sigma(W_{xj}x_t + W_{hj}h_{t-1} + b_j) \\ c_t &= c_{t-1} \odot f_t + i_t \odot j_t \\ h_t &= \phi(c_t) \odot o_t \end{aligned} \tag{2.4}$$

Where W denotes the weight matrix, $\odot$ element wise vector product, ϕ an activation function (usually $tanh$), σ an activation function (usually the sigmoid function), b biases weights.

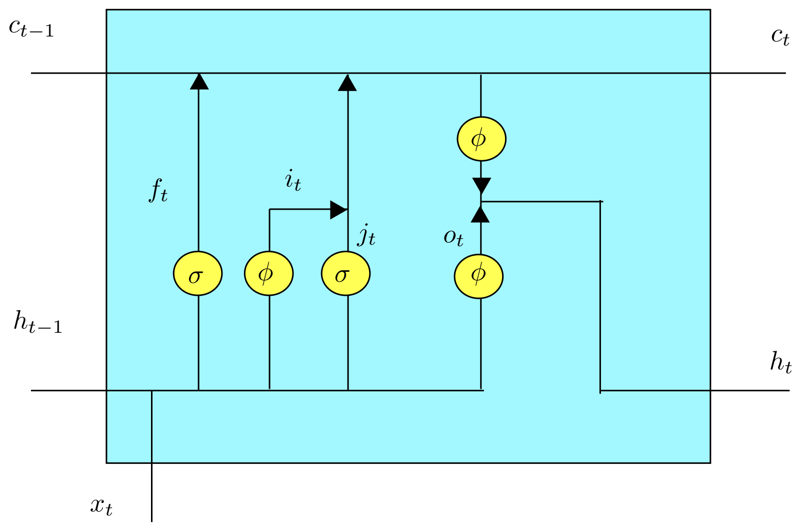


Figure 2.1: Diagram of one LSTM unit

2.2.5 Gated recurrent units

GRUs have been developed in 2014 by Cho et al. [39]. It is a slight simplification of LSTMs. It only uses 2 gates because it removes the output gate. This makes it less powerful but computationally faster. On certain datasets, GRU can outperform LSTM [36].

Terminology replaces the input gate with a reset gate called r. The update gate is denoted by z. The following formulas then represent the gated recurrent unit:

$$\begin{aligned} r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\ z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ \bar{h}_t &= \phi(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \bar{h}_t \end{aligned} \tag{2.5}$$

2.2.6 Temporal Convolutions networks

Temporal Convolution networks (TCN) are a class of network architectures. In Bai et al. [27] they are described by the following two properties. 1) Every convolution has to be casual, that is, no information from future inputs can go to an output in its past; 2) Any length sequence can be used as input and result in the same length output.

Point 1) can be achieved by only convolving an output t with elements from time t and earlier in the previous layer. Point 2 is achieved by the fact that every convolution applied on a 1-dimensional sequence is sliding over an input tensor with a certain kernel weight and applying the dot product to combine them into a single output. To keep new layers the same size as the previous one zero padding of length $kernelsize - 1$ is added.

Figure 2.2shows a diagram of a TCN model. The kernel size determines how many time steps are taken into account when calculating a new output. In the diagram, the kernel size is 2. The channel size determines how many layers there are and is set to 4. To increase the effective history size a model uses a dilation factor which is 1 between the input and the first hidden layer, 2 between the hidden layers and 4 between the last hidden layer and the output. The dilation factor is exponentially increased in the next layer i.e. 2^i where i is the layer number. Each block in the diagram represents a block unit. This unit applies at least one casual convolution. However, Bai et al. use weight normalization, a *rectified linear unit*ReLU and dropout to improve the model in this step. Each step is applied twice to add more weights per block.

TCN models are applied to sequence modelling tasks and outperform recurrent architectures and can be designed to have fewer parameters compared to conventional recurrent neural networks [27].

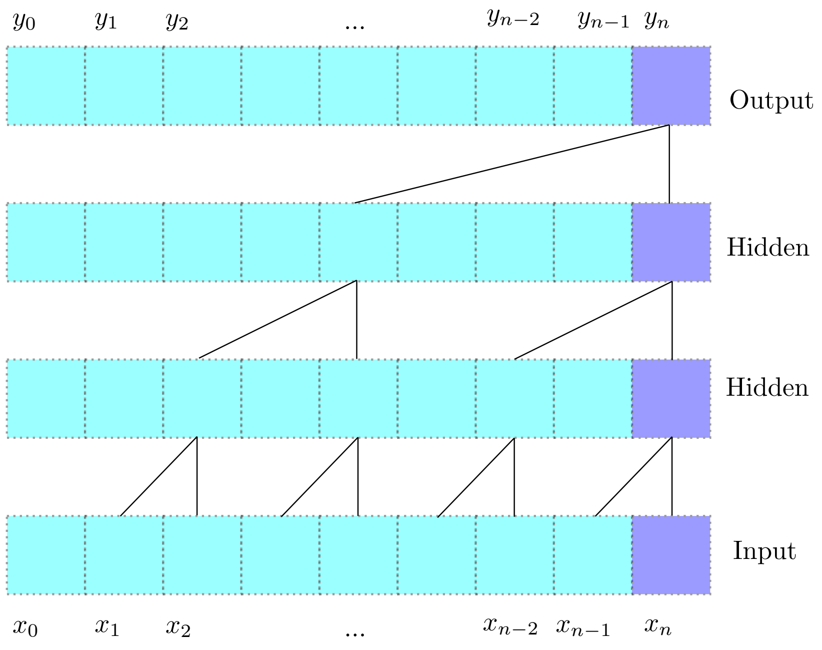


Figure 2.2: Diagram of a TCN

2.3 Real-time system concepts

The CAN bus runs on a real-time system. Buttazzo [40] describes real-time systems as computing systems that must react within precise time constraints to events in the environment. So, not only is the correct operation of the system important but also the timing of the system. The CAN bus implements a fixed priority schedule and is non-preemptive, that is messages have a fixed priority and cannot be interrupted.

The technical committee on real-time systems [41] describes a task as a sequence of instructions that, in the absence of other activities, is continuously executed by the processor until completion. A task is typically activated several times on different input data, generating a sequence of task instances, called jobs. A task can have several activation patterns and the relation between subsequent jobs. There are three different patterns. A periodic task is repeated at regular intervals of time, such that the activation of consecutive jobs is separated by a fixed interval of time, called the task period. An aperiodic task can be activated at arbitrary time intervals. And lastly, a sporadic task has its activation separated by a minimum interval time.

We also deal with start time or release time jitter. It is the maximum deviation of the start time of two consecutive instances [40].

2.4 CAN protocol

2.4.1 Frames

A *controller area network* (CAN) is a broadcast bus that is widely used in automotive systems. More precisely, it is a multi-master serial bus that connects *electric control units* (ECUs) together. These ECUs (also referred to as nodes) broadcast messages periodically, that is at regular intervals, or sporadic, which is at irregular intervals. This broadcast is then received by any other ECU.

The broadcasts that can be sent are called frames. They are a sequence of bits and can be one of the following types:

i Data frame

ii Remote frame

iii Error frame

iv Overload frame

The data frame is responsible for actually sending and retrieving data. A base format and an extended format exist. This extended version of the specification was released to add more features. Data transmission is automatically performed however, it is also possible to specifically request data by sending a remote frame. An error frame is used to inform the whole network of relevant errors. Lastly, the overload frame is used to add a delay between frames.

The first part of each of the frame types contains a message ID which directly indicates the priority of that message. This priority determines which message is transmitted on the bus first this process is called **arbitration**. The message ID does not have to be unique for an ECU. One ECU can send messages with several different IDs and multiple ECUs can send messages with the same ID. In the base frame, this message ID consists of 11 bits. The priority of a message is decided by the following process, all ECUs start transmitting on the bus synchronously. Each node sends a bit on the bus while also monitoring. In case a node sees a dominant '0' on the bus while sending a recessive '1', it loses arbitration and knows it is no longer the highest priority node. After this process, the node with the highest priority can continuously send data.

Chapter 3

Related work

This section summarizes all found related works. The sections focus on surveys or reviews of a specific field if they exist. These general works often condense the most important information about the subject and contain references to many key papers. Some of these papers stood out in particular as relatable and are discussed in depth. First in Sec. 3.1 *intrusion detection systems* (IDS) are covered because of their similarity in implementing solutions. We then describe an interesting work for fingerprinting drivers in Sec. 3.2. After this inference of properties in Sec. 3.3 is described since extraction of features could be very useful. Forecasting of time series in Sec. 3.4 provides many methods that have been successful in the past for a similar problem. And finally, Sec. 3.5 goes more in-depth on the most promising methods.

3.1 Intrusion detection systems

An intrusion detection system is a system to detect malicious messages on a network. They have become increasingly necessary for the security of real-time systems. Interestingly they often apply machine learning techniques to determine if something is an anomaly, learning what is normal behaviour and predicting what is not. There are similarities to the problem of this paper as the same input data is used. It can give great insight into solving the problem of this paper.

Intrusion detection is a big field with many surveys [42, 43, 44, 45]. We will discuss these surveys in Sec. 3.1.1. This work focuses on in vehicle-network [46, 47]. These will be discussed next in more detail. After that, some of the works will be discussed because of their relevance.

3.1.1 Surveys

Wu et al. [47] made a survey of intrusion detection for in-vehicle networks. Their work provides an overview of experimental attacks on automobiles, one of them being the bus-off attack. State-of-the-art intrusion detection technology is discussed. They distinguish 5 different types, (i) Fingerprints-based (recognition of unique attack characteristics), (ii) Parameter monitoring (comparing parameters of the bus i.e. frequency of transmissions) (iii) Information-theoretic-based (comparing message entropy) (iv) Machine learning-based (classification and deep-learning techniques) (v) Other methods. After this, they compare twenty intrusion detection methods that have been researched (most of which are machine learning-based). Although all of these methods are interesting due to the low error rate of these methods the following two jump out as interesting for this work. That is the work of Marchetti and Stabili [48] who uses machine learning to analyse message ID sequences (so a very similar input as our problem) and the works of Taylor et al. [11] who use RNNs. What is still different from this work is the output of all the works found in the survey. Where the IDS's often only have a single output bit for a detected intrusion.

Al-Jarrah et al. [46] investigate several different intrusion detection systems for vehicle networks, not just limited to the CAN bus. First, they provide an overview of the intra-vehicle networks. After this, they discuss the advantages and disadvantages of commonly used networks. They reviewed and categorized 42 works of research on IDS and provide detailed comparisons looking at features and feature selection methods, evaluation data, performance metrics and benchmark models, and targeted attack types. Furthermore, a summary of IDS systems is given. Lastly, they discuss the challenges and future of the subject. They note that several works have used machine learning from the methods they compared. Especially supervised algorithms were present a lot, but they require labelled data which has a high cost to produce and requires domain knowledge. There is limited use of unsupervised and semi-supervised in intra-vehicle IDS. This work aims to use supervised algorithms. The use of labelled data is discussed in chapter4.

3.1.2 ID Sequence

One of the particular interesting works on IDS systems is based on message ID sequences, that is a trace of IDs in chronological order, similar to the input in this work. Marchetti and Stabili [48] based their IDS on the legality of message ID sequences in CAN data. The goal is to identify injection-based attacks, that is, an attack where forged CAN messages are inserted into the CAN traces. To detect an intrusion, they propose to create a model based on the normal behaviour of a CAN network. This is done by looking at recurring patterns of message ID sequences. A matrix of true and false values can keep track of all legal (normal operating behaviour) transitions from one message ID to another. This matrix is trained on actual data. An advantage of this method is the computational cost, which is very low as only one lookup and one comparison are needed. Their results are promising. Basic injection of new messages is detected 100% if the number of injection messages per second is more than one. Realistic attacks are tested by simulating an attack scenario. Three types of attacks are used, 1) Replay 2) Bad injection 3) Mixed injection. Experimental results show that attack 1 and 2 are almost always caught and attack 3 in around 30% of the cases. Finally, the method has a low false-positive rate. The paper concludes by arguing that future work includes the integration of the algorithm and multiple features that could be used for an actual IDS system. The similar problem input to this work and the low computational cost of the solution shows the viability of making predictions.

3.1.3 LSTM

Taylor et al. [11] proposed an LSTM for CAN bus anomaly detection. For each message ID they encountered a separate network was trained by looking at captured car data. The input consists of a sequence of CAN messages, the bits out of which the messages consist are directly used as input vectors. Each network consists of two hidden layers, two LSTM layers and an output layer. Through experimentation, the architecture was selected. Only one bit is used as output indicating the likelihood of an anomaly for that particular message ID. If attack messages (messages changed to maliciously influence the CAN bus) differ in their bits a lot from normal messages the likely hood of anomaly goes up. Their work depends on the messages having a predictable structure of bits. The method is tested without looking at actual attack data since as they state, no public repository of attack traffic is available. Different types of synthetic anomalies were created. The detection accuracy compared to the false positives is high in almost all cases. They note that for certain message IDs it is way harder to predict the bits of a message. The fact that for each message ID the sequence is treated independently is a drawback. It results in high computational cost and certain connections between message IDs can never be learned. Again, the authors argue that a practical anomaly detector would require multiple sensors or features to work.

In both works of Pawlec et al. [12] and [49] neural networks, particularly LSTMs, are used to predict the contents of CAN messages. Looking at irregular data fields of these messages they use their methods to detect intrusions. Both works rely on the assumption that there is a dependency between previous and future data fields in automotive CAN. An assumption which also required in this work.

3.2 Fingerprinting

Xun et al. [50] proposed constructing fingerprints of an automobile driver. Motivated by security in modern vehicles they developed a technique to accurately identify drivers from just CAN data. The technique works by combining a *convolutional neural network*(CNN) and a *support vector domain description*(SVDD) [51] a statistical method for outlier detection. The CNN first processes several features extracted from the CAN data. The output can be used to identify drivers. This method was tested on two different cars with 15 drivers. On one car they reached 98.216% accuracy and the other 98.644% identifying the drivers with just the CNN. They used the SVDD model with the output of the CNN to determine if the driver is not authorized to drive that car. The accuracy of illegal driver detection reached 95% and 98.9%. The models are trained and tested on normal hardware including a NVIDIA GTX 1660 TI. Still evaluating a single test sample took between $0.2547ms$ and $0.3244ms$. The work shows how machine learning models are applied to CAN data and can efficiently identify drivers.

3.3 Inference of properties

Real time-systems often consist of tasks with certain parameter information. These properties can be inferred by analysing an execution sequence. The inferred properties could be very useful in predicting the next tasks or messages.

One of the most important properties for predicting what happens next is the task period. Several works have worked on inferring it [13, 3, 14].

Liua and Yang [13] showed how to make a side-channel attack more practical. Without the need for previous information the attack is more realistic to execute. They achieve this by inferring task parameters by analysing the execution sequence. Their method constructs a set of candidates for task periods. The candidates are then selected using *discrete Fourier transform* (DFT) and one method the authors made themselves. After this, the execution time of each task is inferred by a series of optimization problems assigning for each possible period candidate a chance of having that period. Their method is tested and results show it works successfully in most cases. The work shows the potential of inferring properties.

Vădineanu and Nasri [3] have introduced a period interference framework based on regression-based machine learning. Their tool can help with the development, anomaly detection and diagnosis of real-time systems by inferring periods of tasks. The tool uses a system output trace which means it can be applied to any system with a single processing resource. They explain binary traces, a trace that shows when a task is occupying a resource, and ternary traces, a binary trace including idle times. They show how one can obtain them from a CAN bus by only observing messages. The technique first extracts features from the traces. After this, it applies regression-based machine learning (RBML) to get to a final output of periods. The paper compares many different regression methods in both automotive benchmark applications and synthetic traces. Their results have a very low error of only 1.7% on real data.

Period interference on the data can be used to more accurately predict the next messages. This could be done by directly using the period as a feature in a machine learning algorithm. Furthermore, the binary and ternary traces which are explained could also be used as input for predicting single task IDs. Binary traces, ternary traces and variations could be beneficial for prediction as much perhaps unnecessary data is filtered out. It might be promising to generalize their research to be able to predict future events.

3.4 Forecasting of time series

Forecasting of time series is a well-researched field with many surveys. The use cases include weather, financial markets [52] and population forecasting. The goal is to predict what will happen in the future to anticipate and prepare for scenarios. Therefore, it is widely investigated. The comparison of methods could be useful to determine what methods are suitable for this work. Unfortunately, most of these methods are not directly applicable to the sequence modelling problem this works deals with.

The Makridakis Competitions are open competitions to evaluate and compare the accuracy of different methods, the most recent comparison being the M4 [15]. Furthermore 3 surveys include the more general work from Parmezan et al. [5] and Torres et al. [16] and Lim et al. [17] who all focus on deep learning methods. On top of this, there are numerous surveys that focus on individual methods like Fuzzy time series [18], Support vector machines [19]. All these works aim to figure out the best methods or algorithms for forecasting time series.

Parmezan et al. [5] have evaluated many statistical and machine learning models for time series prediction in their survey. It contains a detailed explanation of all algorithms most notably artificial neural networks and ARIMA. It is continuous by formally defining time series. Two ways of looking at the prediction problem are described i.e. global and local methods. All algorithms are applied to real-world and synthetic data sets. The results are then compared, there is no clear winner as many algorithms are better at certain data sets than others.

Torres et al. [16] present, in their work on deep learning for time series forecasting, a mathematical formulation of the time series problem. All deep learning architectures typically found for time series forecasting are discussed and explained. Practical aspects are compared and related work is summarized in several fields. Their work does not actually apply the methods to compare results however, it can serve as a detailed reference for deep learning methods and contains a lot of practical information. Unfortunately, their work does not analyse or use sequences. The predicted values are not discrete classes. With enough pre-processing on data one can convert CAN bus data to a relevant time series problem and therefore apply the techniques described. However, information would be lost and the end results unpredictable.

3.5 Sequence modelling and NN

This section is aimed to collect many explored works that employed neural networks for sequence modelling and making predictions. In particular, LSTM networks have been used for a variety of sequence modelling problems. For example, sensor prediction [20], trace restoration [21, 22], emergency event prediction [23]. Others gave improvements for learning LSTM [24] or a complete review [30] and applied architecture search [25]. Others used TCN for critical care prediction [26] or showed that TCN outperform RNN [27]. RNN themselves have also been reviewed in their capabilities [28, 29].

Casagrande et al. [20] proposed a solution for predicting sensory events in smart homes. These smart homes are equipped with motion, magnetic and power sensors to identify activities throughout the house. They gathered real-world data from the sensors to predict when certain activities occur next. Probabilistic methods in combination with LSTM networks yielded the best result. As the problem they solve for predicting sensor events could be viewed the same as predicting messages the paper is relevant. The biggest difference is that their input is binary sensor data, the data is composed of whether or not a certain sensor is triggered, CAN messages are far more complex than this. However, by simplifying the CAN data it would be possible to overcome this difference. If IDs in a CAN trace are seen as a binary event that are assigned the value '1' when it sends a message and '0' otherwise, their solution [20] can be used. Furthermore, Casagrande et al. [20] not only predict what event(sensor) occurs next but also at what time it occurs. This is done by adding more output classes to the network which each represent a certain time window. These windows are rather limited in that they cannot be made more precise without a more complex network. The input encoding they use is one-hot encoding.

Sucholutsky et al. [21] describe a method where an LSTM network is used to restore system traces, in particular, CAN traces. They argue that clean data is a crucial component of numerous machine learning algorithms. However, real-world data is often contaminated with noise, loss and other imperfections. To fix this, their approach can predict or restore the missing data.

To predict specific events arbitrarily far into the future Sucholutsky et al. [21] tried to increase the size of the output layer by the number of events they desired to look into the future. However, this leads to lower accuracy the further they predicted and new training was necessary every time they changed the desired distance. Instead, their method can look arbitrarily far into the future by reusing the output event in a new step as the last event that occurred. This can be repeated for how long is needed. They found that even when the model makes a mistake it is robust enough to continue predicting correctly afterwards. Their method is general as events are one-hot encoded and accuracy is high. It does require pre-compiling dictionaries on the trace data which means a dictionary of events is necessary. This dictionary contains all states and their transition frequencies found in the training data. They give several ideas for improving their method. For example, for a more accurate restoration of a message the past and future messages could be used (their method only looks at previous messages). This requires bi-directional LSTM like Zhou and Huang [22] did in a similar paper on recovering missing sensor data. Lastly, Sucholutsky et al. [21] point out that a method where an encoder network is used could solve the issue of having to pre-compile data.

In summary, the work of Sucholutsky et al. [21] serves a different motivation as our work. The steps they take to predict events for restoring data are similar to what this work is trying to achieve. For this reason, the approach to the solution and the potential improvements they point out are considered in this work.

Zhao [53] provides a survey on event prediction in the field of big data. He summarizes existing techniques and application domains, provides standardized evaluation metrics and procedures and discusses the current state of research. His work also concludes that due to models becoming ever more complex they become too complicated to be interpreted by humans, raising serious challenges since raising trust in a model becomes more difficult. For some domains like medical or law, this is especially important. Furthermore, ever more complex networks become more vulnerable to noise and attacks as they can easily be manipulated. This is interesting for a possible counter defence when this work is used in conjunction with an attack. Since adding noise to messages or the CAN bus can then easily fool an attacking entity.

Jozefowicz et al. [25] applied architecture search to check if LSTM are the best models or if improvements to the current architecture exist. Their findings are that GRU are better in most cases unless the forget biases for LSTM are set to 1. Almost all final networks that were found looked like a GRU.

Chapter 4

Solution approach

By now all details about the problem, the involved technology and related works should be clear. This section will go in-depth about the solution and how it is constructed. First, an overview of the problem is given and why machine learning is chosen. Then the availability of data is discussed in Sec. 4.2. After this, how we deal with the input for machine learning is discussed in Sec. 4.3. All potential machine learning algorithms are listed in Sec. 4.5 and in Sec.4.6 their implementation is talked about. And finally what tools we use in Sec. 4.4.

4.1 Overview

As a reminder, the most basic problem we aim to solve is that of predicting CAN messages. So given as input a time series of previous message ID values with a window of length w, predict which message ID appears on the bus next. We also take into account timing information, the time between the start of a message and the next one to appear on the bus.

Several applications where machine learning on CAN data sets is successfully applied are listed next. First, Xun et al. [50] proposed constructing fingerprints of a driver with a high success rate. This technique allows recognizing individual driver behaviour based on just the messages in a CAN bus. It shows how CAN data is perceptible to machine learning. As discussed in Sec. 3.2.

Secondly, IDS based on machine learning is [49, 12] discussed in depth in chapter. 3. However, these works do not generalize the prediction problem and often result in just one binary output (i.e. is a certain driver controlling the car or is a message an intrusion).

Given the success of machine learning in time series prediction and IDS systems we conclude, that if trained and constructed properly, machine learning will be able to solve the prediction problem with reasonable accuracy.

The aim of the solution is not just to find a correct solution but also to make it efficient and fast enough to predict before a next message is observed. We want the solution to have a low memory footprint because then it is applicable to embedded systems with limited resources and memory. This will result in a Pareto plot of solutions existing where accuracy can be traded for faster run time. Any sufficiently working solution is compared to others.

4.2 Availability of data

An often occurring problem for machine learning projects is that there is not enough, or high enough quality, data available. However, looking at how much data other works like Xun et al. [50] need, the amount of data available for CAN busses is more than enough for successfully applying machine learning. Three sources of data are now discussed.

4.2.1 Existing CAN data sets

For an actual data experiment, we use an automotive CAN data set created by Dupont et al. [54]. From here this data set is referred to as the real-world CAN data. This dataset contains several traces of CAN data from two cars as well as a simulated prototype. It is originally meant for IDS. However, it contains raw unmodified data which we need for this work. The traces we use in this experiment are thus free from attacks and come only from one car. A single trace consists of more than 16 million lines of CAN events, these also contain timestamps. More information and statistics on this data set can be found in Sec. 5.10. Using the real-world CAN data presents a way to evaluate performance not only in the accuracy of models but also lets us evaluate how efficient a model is in a practical scenario.

4.2.2 Generating data

Furthermore, for different experiments, we use synthetic data. In this case, we opted to use a synthetic trace generator from the Python library Simso [55]. This trace generator contains additions made by Şerban Vădineanu [3] in order to use a non-preemptive execution model. The advantage of a generator is much finer control of what the data will look like. It produces a data set according to the following parameters which are explained after.

1. Type of dataset (Automotive)
2. Dataset size (default = 50)
3. Number of message IDs (Varied in experiments, default = 20)
4. Utilization (Varied in experiments, default = 0.5)
5. Alpha (0)
6. Jitter (0)
7. Preemptive (Non-preemptive)

The 'type of dataset' affects how task periods are assigned to each ID. For 'automotive' this is determined by a model from Kramer et al. [56]. They provide a distribution model which includes common periods used in engine management systems and how often each periods occurs as a share of the total. For each period {1,2,5,10,20,50,100,200,100}ms a respective share of {3,2,2,25,25,3,20,1,4}% is used. When generating a task a random period is chosen depending on the share.

The 'dataset size' determines how many traces are generated by the tool, this means with a lower 'utilization', less data points are generated as task IDs do no switch as often. To always include enough task IDs we adjust the 'dataset size' to provide enough data points for the experiment.

We experimented with the number of message IDs and utilization. We did not use the alpha(execution time variations) and the release jitter in the experiments. Lastly, as the CAN bus employs fixed priority scheduling and is non-preemptive we use a non-preemptive execution model.

4.2.3 Constructing data

It is also possible to construct data by monitoring the CAN bus in a car. Any car can be used for this purpose, by hooking into the *on-board diagnostic* (OBD). Each detected message can be represented in the same format as described in the real-world CAN data set. In this work, we did not experiment with our own data.

4.3 Machine learning input

The input data needs to be encoded into a form that is accepted by machine learning models. Since the input to a machine learning model is a tensor all the message IDs need a conversion to a tensor. Although it is potentially possible to let the algorithm run on any encoding it is way more efficient to already transform the input into a logical form. An algorithm can then learn the features of this encoding.

The 11 bits for a CAN bus message ID can lead to 2048 different classes. It is beneficial to normalize IDs to reduce the number of classes and therefore the size of the models. So instead of using CAN message IDs directly, they are normalized. This process also helps with encoding data as the reduced number of classes allows for one-hot encoding.

One-hot encoding is common in machine learning. A class can be represented by a vector of '0's with only at the position of the class a '1'. Leaving for example message IDs as numbers will result in the unwanted consequence that numbers close to each other are related. Furthermore one-hot encoding has the advantage that we can easily convert any natural number to a float tensor. Several machine learning frameworks and algorithm implementations only support float operations.

The normalization process goes as follows, the message IDs in the data sets are first collected in a big sequence of 85 unique IDs. Normalization happens in order of priority. The lowest message ID numbers, which are the highest priority, are converted to the lowest numbers 0 to 85. Before being passed on to the model the list of message IDs needs to be encoded, the experiments use one-hot encoding. In other words, each ID gets a unique vector with length of 85 where all values are set to 0 with only the index of the ID set to 1.

After the normalization and one-hot encoding process, the tensors are then grouped into windows of a certain length w, each group has the next one-hot encoded message ID as its label.

We experimented by adding timing information, the time between two messages, to the solutions. CAN data sets are composed of IDs but also contain a timestamp for each message. We use these to calculate the time between a message and a the next message in microseconds. The input vectors of the network are extended in several different ways to include this information. First, the timing information was added directly as a float. This is done by appending the input tensor group with the full time value. A different method prepends this value instead of appending it. Because the message ID values in the input tensor are either 0 or 1, large time values can have disproportional effects. Therefore, another approach is also used. In the work of Casagrande et al. [20], the authors use several different classes to also predict time using an LSTM. To apply this concept, we added several different one-hot encoded classes to the input vector. The method we named 4C adds 4 new classes after the one hot encoded IDs. The 85C method doubles the size of the input vector adding 85 distinct timing classes. In order to convert the time values to each class, we divided each time value by the average time divided by the number of classes. Several discrete time zone classes. Unfortunately, this adds complexity and is limited in precision as the more classes are added for higher precision the models get larger and thus slower.

4.4 Tools

In order to collect data, convert data to suitable formats and apply machine learning algorithms Python is used. Python as a programming language offers fast development time, is often used when dealing with machine learning, and is widely supported on many platforms. The ease of use when working with data is useful as described in Sec. 4.2. Furthermore, many frameworks are available to aid with machine learning that will be discussed in Sec. 4.4.1.

4.4.1 Frameworks

The three most important frameworks are Keras, Pytorch and TensorFlow. All of them provide easy access to state-of-the-art algorithms and often only differ slightly in syntax or features. For the experiments in Chapter 5 we use Pytorch [57].

4.5 Motivation ML-based solutions

4.5.1 Models

One of the most important factors for the success of machine learning is the representation of the input and output data and the architecture of the model. We use 3 different models, listed below, which each have their own advantages and disadvantages. Each selected machine learning algorithm was picked by looking at relevant other works, and their success in other domains.

1. Long short-term networks (LSTM) [38]
2. Gated recurrent unit networks (GRU) [39]
3. Temporal convolutional networks (TCN) [58]

The most promising algorithm from the literature and related work empirically seems to be LSTM networks [38]. They are used a lot in prediction [20, 35] and often generate one of the best results compared to other algorithms. They also perform well in other tasks like speech recognition [7]. Compared to regular neural networks LSTMs are designed to take the temporal component into account. They also deal with the vanishing or exploding gradient problem which is a common problem with regular RNN. A disadvantage is that they are hard to train and require a lot of data.

The more modern variation on LSTM are GRUs. They are slightly simpler and can on occasion outperform LSTM [36]. Architectural search has pointed out that LSTM converge to GRU [25]. Since we aim at designing a solution with low run time and a small memory footprint, and since GRUs and LSTMs are almost equivalent in terms of accuracy, we use GRUs instead of LSTMs if they perform better.

As TCN are a broad class of networks which feature an encoder and decoder part. Their efficiency is a very attractive property for this project. They are considered state of the art in predicting problems [27].

4.6 Implementation

We constructed an LSTM, a GRU and a TCN, these three machine learning models are implemented in a Python environment using the PyTorch library [57]. First we will discuss the implementation of both the LSTM and the GRU. After that the TCN implementation.

4.6.1 LSTM and GRU

The LSTM and GRU are derived from the default model present in the Pytorch library [57]. Besides that, we add an extra linear layer to the two RNNs to go from the hidden layer to the output layer size and to improve accuracy. The hidden layer and the output layer can be the same size, and for most experiments this is the case.

At first this extra linear layer was only added to convert layer sizes but through testing we found it improves accuracy. We attempted to improve accuracy even further by adding multiple linear layers to make the network deeper. This did not increase accuracy and only led to a greater model size and training length.

We initialize all hidden cells to zero and we set dropout to 0.1 to prevent over-fitting. The input and output size of the networks are set to the number of message IDs we use in the data set that is used. The *hidden_layer_size* and the *layer_size* vary depending on the experiments but we set them by default to respectively 85 and 1.

4.6.2 TCN

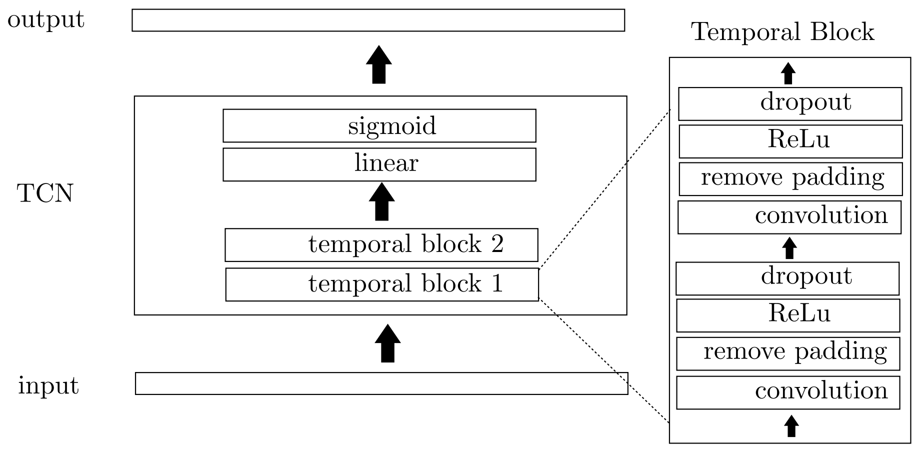


Figure 4.1: Diagram of the TCN implementation

The TCN model is derived from the implementations used by Bai et al. [27]. Figure 4.1 shows a diagram of this implementation. The network consists of multiple Temporal Blocks. Each block adds the following steps twice: convolution layer, remove padding, ReLu activation layer, and apply dropout. On the output of the temporal blocks a linear and a sigmoid layer generate the final output.

To construct the network the number for the *channel_size* determines how many Temporal blocks are chained. Each new block doubles the *dilation_size* which is used in the convolutions. It starts at 2. For example, in block 1 we have a *dilation_size* of 2 and for block 2 a *dilation_size* of 4.

We set the input and output size for the convolutional layers equal to the number of message IDs in a data set. The kernel size and the dilation amount are set to the variables *kernel_size* and *dilation_size*. To make sure the layers remain the same size each convolutional layer adds padding equal to $(kernel\_size - 1) * dilation\_size$.

The dropout for the convolutional layer is set to 0.1 but we find that higher values can help when over-fitting is a problem, this happens often when time cross-validation splits the data set in too few parts. We find that using 5 splits is most often adequate to solve the problem. The train window parameter influences this a lot, and increasing the number of splits to 10 or the dropout to 0.2 help to solve over-fitting. Changing the dropout might be preferred as this does not increase training time compared to more time cross-validation splits.

We choose the parameters which determine the number of weights of a network, like the hidden size for the RNNs and the kernel and channel size for TCN, so that the models are comparable. We aim to keep the networks below 100.000 weights to keep the execution time of the network within a millisecond. This leads to values of 2 for the kernel and also 2 for the channel size and by default the hidden layer size is set to 85.

We attempted to find better architectures by adding and removing convolution layers. We concluded that we could not find any improvements. It must be noted that to test the performance of a network a large data set is needed and iterating different architectures takes a long time. The parameters discussed like the *kernel_size* and the *channel_size* provided enough control on the size of the network so for the experiments we use the TCN model as described previously and originally implemented by Bai et al. [27].

Chapter 5

Results

5.1 Overview

In this section, we present a set of experiments that are conducted to show the effectiveness of the proposed solutions and answer the following questions from the research hypotheses. (**1**) What type of neural network may be a better fit for our problem. (**2**) Should we do the feature extraction by ourselves or let the machine learning solution find it for us? (**3**) To what extent is it possible to make an efficient solution for event prediction?

To help us answer these questions the experiments are grouped into a main experiment for each model on a synthetic and a real-world data set. Furthermore, there are several smaller experiments conducted on a smaller data set to investigate the impact of varying parameters. There is one set of experiments to show the effect of adding timing information as a feature. And finally, we experimented with the utilization and number of message IDs using synthetic data.

5.2 Experimental Setup

For the real-world CAN data we have 2 separate files of data. One is used for training and the other for testing. The training data set is further split into one part for training only and the other for validation of the model. Training is done over several epochs and different parts of the data set. After training on a single part of data the performance of the model is evaluated using the validation set, the data directly after the part used for training. This is time cross-validation as discussed in Sec. 2.2.1. The length of the validation data is determined by investigating the data sets as described in Sec. 5.10. Lastly, the test data file is used to get the final performance of a model.

For the synthetic data set we generated a single file with the parameters discussed in Sec. 4.2. A file contains multiple different traces. The first part of this file is used for training and validation using enough traces to get a specified amount of data points for each experiment. The traces at the end of the file are used for testing, making sure that there is no overlap between the training and testing data.

For hardware, an i5-4570 CPU, 8GB RAM and a NVIDIA GTX 960 are used for both training and evaluating the models.

List of set parameters

Network parameters:

1. Window length (default = 64)

Network parameters RNNs:

1. Hidden size (default = 85)

2. Layer size (default = 1)

Network parameters TCN:

1. Kernel size (default = 2)
2. Channel size (default = 2)

Training parameters:

1. Epoch size (default = 5)
2. Time series cross-validation (default = 5)
3. Learning rate (default = 0.01, halved each epoch)
4. Loss function (default=CrossEntropyLoss)

The window length is determined by testing, higher window lengths can improve accuracy. However, it also increases the execution time of the model. We find the default 64 to be a good trade between accuracy and performance for all models. We initially chose the hidden size to match the input and output size of the model for simplicity. Later testing showed that increasing it further only increased execution time without positively influencing accuracy. Increasing the layer size doubles the number of parameters in a model. This doubles evaluation time and also causes exponentially longer training. With an increased layer size, it is no longer possible to efficiently iterate over different models so we keep it to its default 1 value in our experiments. The kernel size and channel size for the TCN model defaults are chosen to keep similar size models for the RNN to compare them fairly. We found a channel size and kernel size of 2 are needed to achieve this. Most training parameters are empirically assigned. The model did not seem to increase with more epochs, the time series cross-validation is kept at 5 as it did not change the accuracy for different values. The learning rate is sufficiently high for the amount of training. As a loss function cross-entropy performed well for all models.

5.3 Metrics

To compare between models an average accuracy metric is used based on the number of times the model correctly identifies the correct message ID. For the main experiment, we investigated this further by looking at the accuracy per message ID with a certain priority and rate of occurrence. Four metrics are defined:

1. $$Accuracy = \frac{\sum_{id=0}^{max\_id} \frac{correct(id)}{occurred(id)}}{n}$$

   Where max_id is the total number of IDs, $correct(id)$ gives how many times that message ID is guessed correctly, $occurred(id)$ how many times it occurred in total. n is the total number of values over all IDs.

2. $$Occurence\_adjusted\_accuracy = \frac{\sum_{id=0}^{max\_id} \frac{correct(id)}{occurred(id)} * \frac{occurred(id)}{max\_occurred}}{max\_id}$$

   Where $max\_occurred$ is the maximum number of times any message ID occurred. This gives a higher weight to message IDs which occur less often.

3. $$Priority\_adjusted\_accuracy = \frac{\sum_{id=0}^{max\_id} \frac{correct(id)}{occurred(id)} * \frac{priority(id)}{max\_id}}{max\_id}$$

   Where $priority(id)$ is the priority of a certain message ID id. This gives a higher weight to message IDs with a higher priority.

4.

$$Priority\_occurence\_adjusted\_accuracy = \frac{\sum_{id=0}^{max\_id} \frac{correct(id)}{occurred(id)} * \frac{priority(id)}{max\_id} * \frac{occurred(id)}{max\_occurred}}{max\_id}$$

Which takes both metrics into account.

All metrics scale between 0 and 1. The three metrics priority, occurrence and both help to compare how different models and datasets respond to different aspects. In the case of priority, a higher number means that it performs better on the higher priority message IDs. For occurrence a higher number means it performs better on IDs which occurs less. And finally, a combined metric takes both into account.

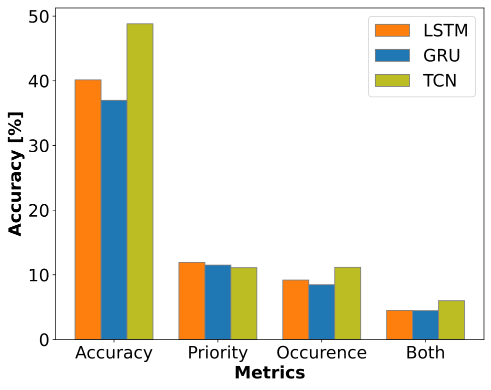
For the synthetic data the ID number does not necessarily determine priority. The sets are generated in a way that the lowest period is assigned to the first ID and this continues until the highest period is assigned to the last ID. The priority metric is therefor not available for synthetic data.

5.4 Main experiment

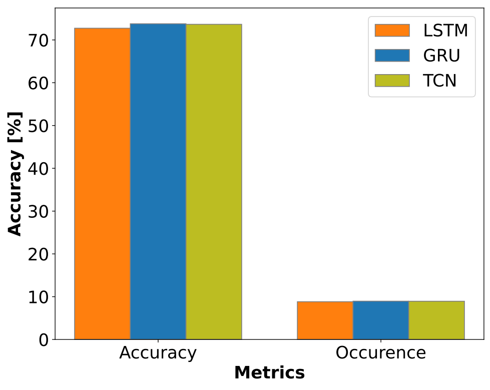
For the synthetic data set we use 100.000 data points for training and 25.000 for validation. We test the experiment with 50.000 data points. For the real-world CAN data set we use the complete data set and again 50.000 for testing.

In figure 5.1 the metrics are compared for each model. In terms of accuracy, the three different models differ slightly when using the real-world CAN data set. This difference becomes way smaller when looking at the synthetic data. In this case, the accuracy is rather high. The TCN responds better based on the occurrence of IDs. In other words, compared to the other models the TCN is better at predicting message IDs which do not occur as often as others. The priority of messages does not seem to affect the performance of the models. So higher priority messages are not predicted any better.

Performance is measured by looking at the average time it took to complete one prediction in testing. LSTM($767.88\mu s$), GRU($708.19\mu s$) and TN($929.69\mu s$). The memory required is measured in the total number of parameters each model contains. For LSTM and TCN this is exactly 65790, for the GRU it is slightly lower at 51170.



(a) Full CAN data set



(b) Synthetic data set

Figure 5.1: Accuracy metrics

5.5 Impact of Parameters

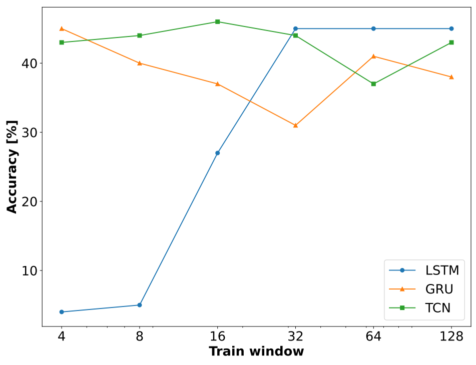
To compare hyper parameters models are trained on 100.000 data points or message IDs for the real-world CAN data set and 50.000 for the synthetic data set. For validation the real-world and synthetic data set use 25.000 and 12.500 points respectively. Time cross-validation is used with $k = 5$, and 1 epoch of training.

In this work, several different important hyperparameters have been identified, most importantly the train window affected several models in different ways, see figure 5.2a and 5.2b. For the LSTM model a higher train window size was needed to work properly. It only makes a few correct predictions with a train window smaller than 16, and it needs at least 32 to compare in performance with the other models. The synthetic data set reflects the same, however, not it is not as noticeable. The TCN model shows a big dip when using the synthetic data. Why accuracy drops sharply at a training window length of 16 and 32 is not clear.

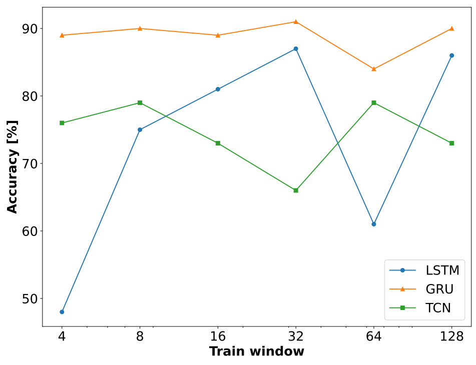
The execution timings of varying training window sizes are graphed in figure 5.2c and 5.2d. An advantage of the TCN becomes immediately clear. Both RNNs have to process each step in the train window in order because the result of the previous cell is needed in the next one. This means that execution time scales linearly with the train window size. For the TCN this is not the case because it uses the same kernel weights and no other input for every step. This results in it being able to parallelly compute the output tensor and a near constant execution time. Unfortunately, only for large window sizes such as 128, do the LSTM and GRU overtake the TCN model.

For the two RNN models varying hidden sizes were tested. This parameter has a big effect on accuracy and performance. See figure 5.2e and 5.2f. It does behave as expected. A minimum hidden size is needed before we start seeing decent accuracy. In this case, around 10 is sufficient. After that the hidden size still has a significant effect however, a higher hidden size does not result in better accuracy. In terms of performance, the networks work faster when using a hidden size close to the in and output size. Which is clearly visible. After this, with higher hidden sizes the execution time also increases.

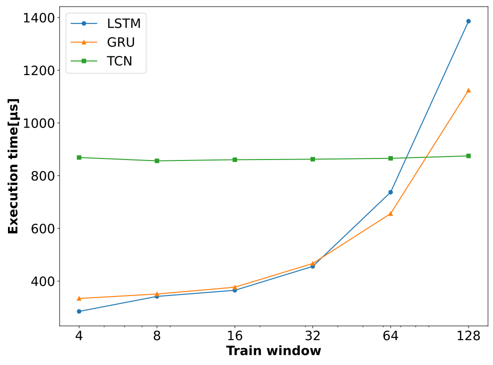
For the TCN, different kernel and channel sizes are experimented with. See figure 5.3a and 5.3b. Both options increase the number of parameters a TCN model has to train. For the kernel size, this has little effect as all calculations can still be made in parallel. For a higher channel size execution time increases linearly by a lot. Accuracy results are more though to analyse however, a small network with a kernel size and channel size of 2 seems large enough for our problem. With a higher channel number of 4 and a kernel size higher than 2, the network becomes too big to train with the current procedure.



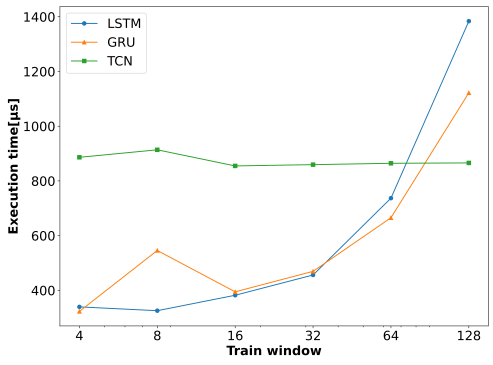
(a) Varying train window CAN data set



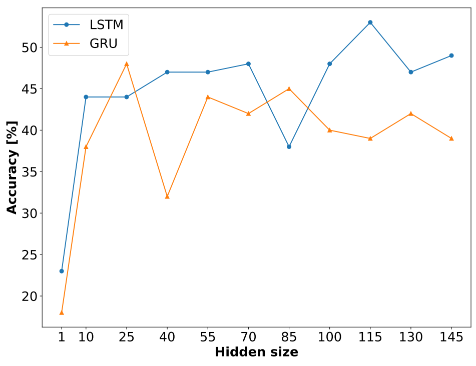
(b) Varying train window synthetic data set



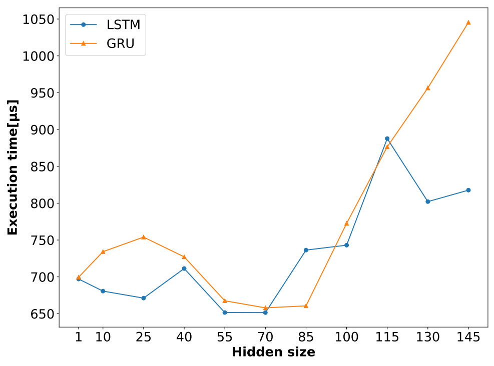
(c) Execution time varying train window CAN data set



(d) Execution time varying train window synthetic data set

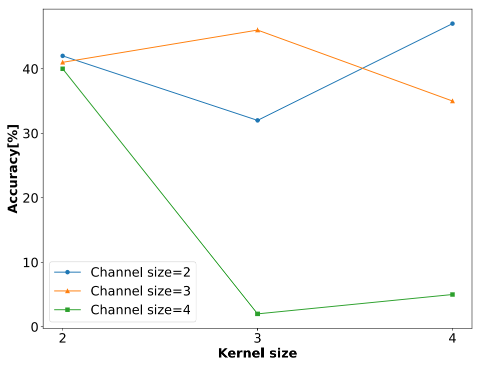


(e) Varying hidden size

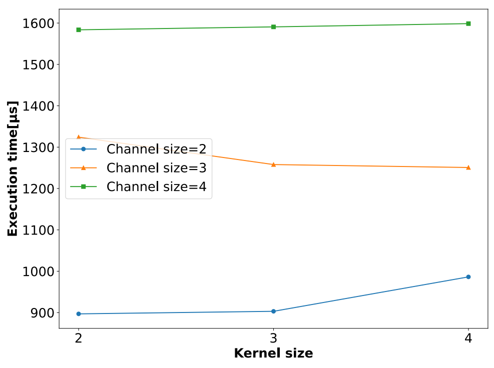


(f) Execution time varying hidden size

Figure 5.2: Varying parameters



(a) Channel and kernel size (TCN)

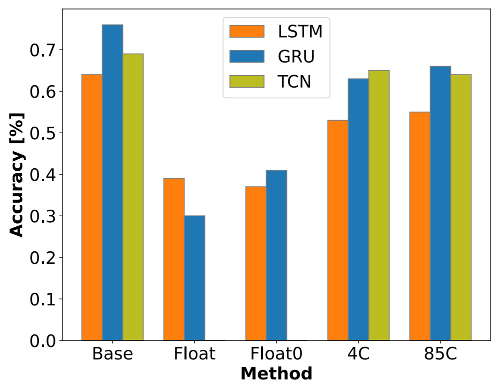


(b) Execution time channel and kernel size (TCN)

Figure 5.3: Varying parameters (cont.)

5.6 Adding timing information as a feature

The experiments which added time are found in figure 5.4. The four methods are Float, Float0, 4C, and 85C as described in Sec. 4.3. Summarized the Float method adds the time values after the input tensor as a float and Float0 does the same but the number gets prepended to the tensor. 4C and 85C add respectively, 4 and 85 classes to the tensor and the time values are converted to these classes. The Float methods add disproportional big values to a one-hot encoded tensor and we suspect the networks are not large enough to compensate for this. Especially for the TCN models where the same kernel is used for both the added time value and the one-hot encoded IDs this leads to the TCN models no longer predicting anything correctly at all. The two class methods performed better but unfortunately not as good as without timing information. Out of all models, the TCN have the lowest accuracy penalty for adding the class time features.



(a) Different methods of adding timing information

Figure 5.4: Added timing information

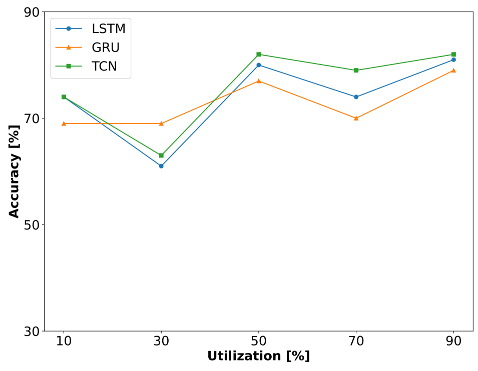
5.7 Utilization

To test the impact of utilization of the bus we varied the utilization of the synthetic data set. The number of message IDs is set to 20, and the other parameters are left at the default as discussed in Sec. 4.2. Figure 5.5a shows that accuracy remains similar depending on the utilization. We hypothesised accuracy would go down at higher utilization rates because the higher utilization the more uncertainties happen in the data. A reasoning behind the fact that high utilization keeps accuracy high is that the biggest change in the data is the inclusion of idle time. The way the models process input is through a window length of IDs. The idle time is not considered anymore and such only the order of IDs matters. The higher the utilization the more examples the models can train on.

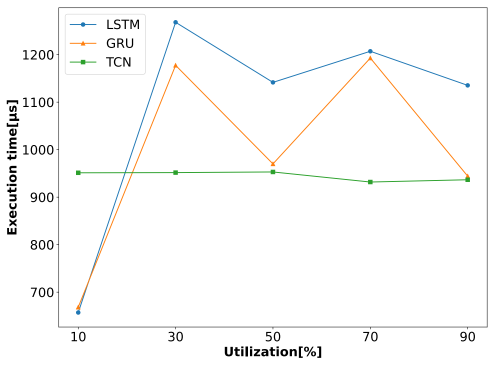
We hypothesize that the big jumps at for example, 30% utilization comes from over-fitting. Throughout testing we often found accuracy outliers with different data sets. Increasing the amount of training data only had a small impact on accuracy. But changing other parameters like the dropout and time cross-validation does help when facing outliers in accuracy. For this experiment we found not better values for the dropout and time cross-validation parameters.

Figure 5.5b shows the execution times. For the TCN the execution time remains stable, this is in line with expectations as the size of the model does not change. The execution time of the RNNs is sporadic, this is partially due to the number of message IDs used in the datasets, as this is set to 20 random variations in execution speed have a higher impact than when using a higher number of message IDs.

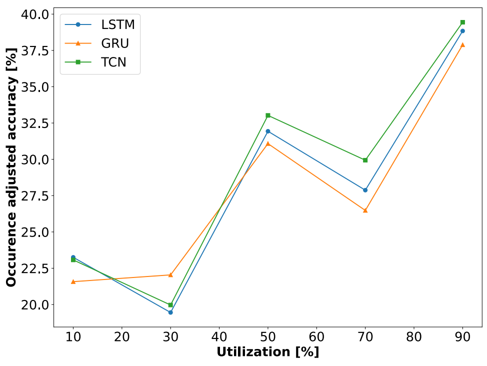
Finally, Figure 5.5c shows the occurrence adjusted accuracy metric. We observe a general trend upwards compared to the normal accuracy. This indicates that with a higher utilization the models get better at predicting message IDs which do not occur often. Between the different models no difference is noticed in respect to their performance on less occurring message IDs.



(a) Varying utilization



(b) Execution time varying utilization



(c) Occurrence metric varying utilization

Figure 5.5: Effect of utilization on the synthetic dataset

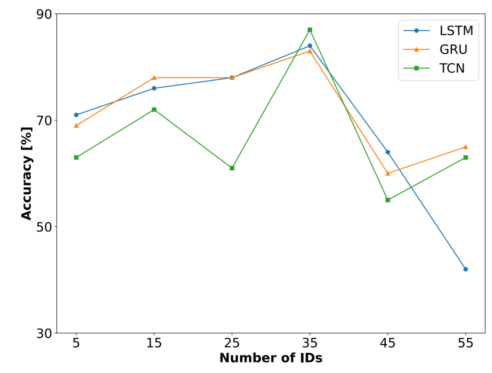
5.8 Number of message IDs

To test the impact of the number of message IDs the bus we varied this parameter while leaving the utilization at 50%. Again the other parameters are kept at default as discussed in Sec. 4.2. figure 5.6a shows that for a very small number of tasks the accuracy is not high, in fact the highest accuracy comes where there are 35 IDs. The TCN model is impacted the most by the change in number of message IDs, as those results are the most sporadic.

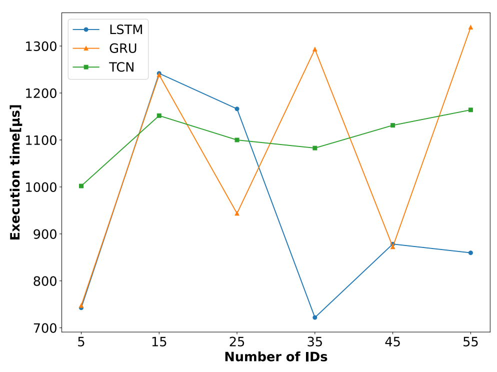
It must be noted that changing the number of message IDs also changes the size of the models by a lot. In figure 5.7 the number of parameters is plotted. For example, the for 5 messages IDs the number of trainable parameters is between 294 and 378 depending on the model while for 55 message IDs the number is between 22344 and $25,536$. By coincidence the TCN and LSTM model have the same number of parameters in this experiment. The GRU has slightly less as expected since it is a simplification of the LSTM. For all models there is a clear exponential growth. We hypothesize the number of message ID causes variance in the results due to the different numbers parameters.

The TCN again is far more stable in execution time compared to the RNNs. Since the whole model changes a lot depending on the number of message IDs parameter, it makes sense to see that the TCN does get influenced, and a slight upwards trend is present. Many of the calculations are made in parallel and we hypothesize that variations are attributed to other variables that impact execution times such as, caching behaviour.

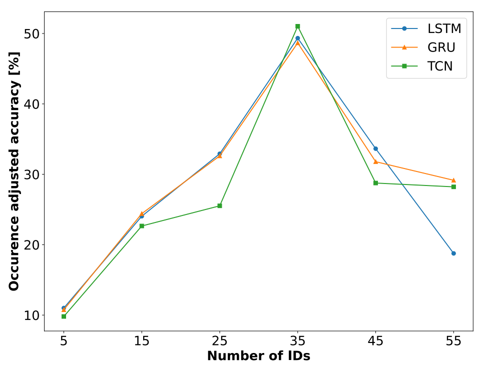
The occurrence adjusted measurement follows almost the same curve as the normal accuracy. This indicates that for a high number of IDs we only observer a small increase, in the performance on message IDs that do not occur often.



(a) Varying number of message IDs

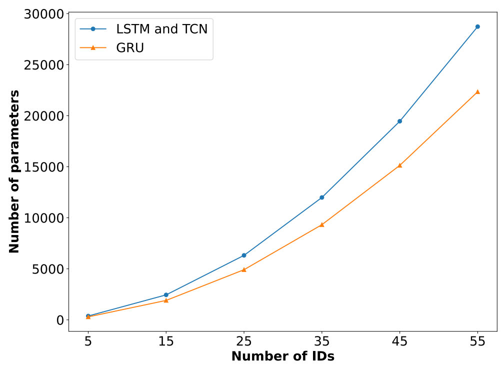


(b) Execution time varying number of message IDs



(c) Occurrence metric varying number of IDs

Figure 5.6: Effect of parameters on the synthetic dataset



(a) Number of parameters for number of message IDs

Figure 5.7: Effect of number of message IDs on network size

5.9 Other experiments

Other encodings were also experimented with like the nn.Embedding option in PyTorch where the vectors can be learned much the same way a neural network can. In this case, the encoding acts like a lookup table that matches a message ID with a unique vector. These are often randomly initialized and then improved once training starts. Unfortunately, it became quickly clear that this only added training time without showing any advantages.

Furthermore, we experimented by using larger batch sizes in our models. By default, we use a batch size of 1 for training and evaluation. Smith et al. [59] show that it is better for Adam, which we used as a training optimizer, to increase the batch size instead of the usual practice of decaying the learning rate. They obtain the same learning curve while using fewer parameter updates which allows for more parallelism and therefore shorter training time. Unfortunately, we observed that some batch sizes decreased accuracy without a clear relationship between the number and the drop in accuracy. We hypothesize this occurred because of the structure of CAN data. Also, no parameters were found that improved accuracy compared to using a single batch size. Increasing the batch size also increases the model parameters and although we can do more work in parallel this is not useful when only a single prediction is required.

5.10 Analysis of data sets

To compare against the execution timings of the models the real-world CAN data set is investigated. On average the time between messages or IDs is $514\mu s$. The percentage of messages which are faster than $100\mu s$ were only around 2%. The distribution of message IDs in the CAN data set does change significantly anymore after 25000 data points. For this reason, the size of the validation set is set to this number.

Chapter 6

Conclusion

6.1 Overview

In this work, we explored how well machine learning can be applied to the problem of predicting real-time system traces, specifically, automotive. The three most promising models were implemented and extensively tested. Results reach almost 50% accuracy on real-world data. Considering the low execution time and small memory footprint a practical application can apply our solution. Empirically many attempts were made to increase accuracy with mixed results. We added timing information as an input, tried batch learning, embedded encodings, and changed architectures, by among other things, including more layers and increasing the layer size. Eventually, there is not a single best model, mainly because varying parameters affected accuracy.

6.2 Answers to research question

1. What type of neural networks may be a better fit for our problem?

   Based on the literature and previous work in predicting sequences we identified three classes of models suitable to solve our problem: LSTM, GRU and TCN. From our main experiment on the CAN data set we conclude that TCN models give the best performance in terms of accuracy. However, the difference between the other two models is very small. Changes in certain parameters of the problem also have a great effect on the accuracy so TCN are not always the best choice. It should also be noted that out of all models the most time was spent on optimizing TCN. Another edge for the TCN, is that execution time is far more stable depending on parameters. Especially changing the train window size shows that for higher numbers the TCNs ability to process data in parallel is an advantage.

2. Should we do the feature extraction by ourselves (using expert knowledge) or should we let the machine learning solution finds that for us? In the former case, a follow-up question is what features should we use? and in the latter case, the follow-up question is how to apply the neural network technique to a problem with a variable input size?

   Adding time as a feature noticeably worsened accuracy. It is likely the timing information is not a very useful feature for predictions. Furthermore, using an embedded encoding in the model was only followed by worsened accuracy and increased model complexity. The normalization procedure applied in this work worked well. Conveniently the models used in this work all support variable input size. Often seen as one of the advantages of the used models.

3. To what extent is it possible to make a solution perform in real-time?

   All experiments are performed on normal consumer hardware as described in Sec. 5.2. From the data set $514\mu, s$ was needed on average to send a message. Slightly lower than the execution times of one prediction for the main experiment. With timings between $700\mu s$ and $930\mu s$. With better hardware or a lower window size for the RNNs, this can even be matched with the time it takes to send a message on the bus and thus predict before the next message starts transmitting.

6.3 Suggestions for future work

Adding timing information to improve the prediction accuracy was not successful in this work. Several different methods such as adding different numbers of classes to one-hot encoding were attempted but some remain untested. One of these untested methods would be to use, in the case of an LSTM, two different LSTM models and merge the output afterwards. One model would just be concerned with timing information while the other uses the data IDs like normal. After this, another model could combine both outputs to determine a final output message ID. A disadvantage of this approach is added complexity and increased execution time as the information of both models would have to be merged by another layer but if adding an extra feature is successfully it could also open up a road to adding even more features. Remaining seems the extra data contained in the CAN messages. We hope these extra ideas can push the accuracy up further.

Much of the time in this project was spent on tuning hyperparameters. Once a possible architecture for a model was made, it took a long time to understand the performance for it. All parameters needed to be tuned first. In hindsight, hyper-parameter optimization in a structured way would have helped a lot. Any future work should look into automating the process with, for example, gradient-based optimization or grid search to iterate on models faster.

This work did not investigate how effective models are at predicting further into the future. If there is a practical need to predict multiple messages into the future this is possible in two ways. The first method is to append the output to the input tensor and remove the first element so the tensor remains the same length. This method is simple to implement it can use the same trained models that exist already, no larger models are required. One downside is that it will require evaluating the model for every extra prediction one wants to make. Errors will also compound since any mistake will be put into the model again. The second method is to train the network from the start to include more predictions. Training will take more time and the further into the future one aims to predict the larger a model will likely need to be to get good accuracy.

Bibliography

[1] J.C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, 2002. 1

[2] Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 103–116, 2019. 1

[3] Şerban Vădineanu and Mitra Nasri. Robust and accurate period inference using regression-based techniques. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 358–370, 2020. 2, 3, 14, 18

[4] Spyros Makridakis, Evangelos Spiliotis, and Vassilis Assimakopoulos. Statistical and machine learning forecasting methods: Concerns and ways forward. *PLoS ONE*, 13, 03 2018. 2, 3, 4

[5] Antonio Rafael Sabino Parmezan, Vinicius M.A. Souza, and Gustavo E.A.P.A. Batista. Evaluation of statistical and machine learning models for time series prediction: Identifying the state-of-the-art and the best conditions for the use of each model. *Information Sciences*, 484:302–337, 2019. 2, 3, 4, 6, 8, 15

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. 2

[7] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 38, 03 2013. 2, 20

[8] John Hawkins and Mikael Bodén. The applicability of recurrent neural networks for biological sequence analysis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(3):243–253, 2005. 2

[9] Razvan Pascanu, Tomas Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. *30th International Conference on Machine Learning, ICML 2013*, 11 2012. 2

[10] Wufei Wu, Renfa Li, Guoqi Xie, Jiyao An, Yang Bai, Jia Zhou, and Keqin Li. A survey of intrusion detection for in-vehicle networks. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):919–933, 2020. 3

[11] Adrian Taylor, Sylvain Leblanc, and Nathalie Japkowicz. Anomaly detection in automobile control network data with long short-term memory networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 130–139, 2016. 3, 12, 13

[12] Krzysztof Pawelec, Robert A. Bridges, and Frank L. Combs. Towards a can ids based on a neural network data field predictor. In *Proceedings of the ACM Workshop on Automotive Cybersecurity*, AutoSec '19, page 31–34, New York, NY, USA, 2019. Association for Computing Machinery. 3, 13, 17

[13] Songran Liu and Wang Yi. Task parameters analysis in schedule-based timing side-channel attack. *IEEE Access*, 8:157103–157115, 2020. 3, 14

[14] Oleg Iegorov, Reinier Torres, and Sebastian Fischmeister. Periodic task mining in embedded system traces. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 331–340, 2017. 3, 14

[15] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. The m4 competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, 36:54–74, 2020. 3, 15

[16] José F Torres, Dalil Hadjout, Abderrazak Sebaa, Francisco Martinez-Alvarez, and Alicia Troncoso. Deep learning for time series forecasting: A survey. *Big Data*, 9(1):3–21, 2021. 3, 6, 8, 15

[17] Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194), Feb 2021. 3, 15

[18] Mahua Bose and Kalyani Mali. Designing fuzzy time series forecasting models: A survey. *International Journal of Approximate Reasoning*, 111:78–99, 2019. 3, 15

[19] Nicholas I. Sapankevych and Ravi Sankar. Time series prediction using support vector machines: A survey. *IEEE Computational Intelligence Magazine*, 4(2):24–38, 2009. 3, 15

[20] Flávia D. Casagrande, Jim Tørresen, and Evi Zouganeli. Predicting sensor events, activities, and time of occurrence using binary sensor data from homes with older adults. *IEEE Access*, 7:111012–111029, 2019. 4, 15, 19, 20

[21] Ilia Sucholutsky, Apurva Narayan, Matthias Schonlau, and Sebastian Fischmeister. Deep learning for system trace restoration. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019. 4, 15, 16

[22] Jingguang Zhou and Zili Huang. Recover missing sensor data with iterative imputing network. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence*, volume WS-18 of *AAAI Technical Report*, pages 209–216. AAAI Press, 2018. 4, 15, 16

[23] Bitzel Cortez, Berny Carrera, Young-Jin Kim, and Jae-Yoon Jung. An architecture for emergency event prediction using lstm recurrent neural networks. *Expert Systems with Applications*, 97:315–324, 2018. 4, 15

[24] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1171–1179, Cambridge, MA, USA, 2015. MIT Press. 4, 15

[25] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on International Conference on Machine Learning - Volume 37*, page 2342–2350, 2015. 4, 15, 16, 20

[26] Finn Catling and Anthony Wolff. Temporal convolutional networks allow early prediction of events in critical care. *Journal of the American Medical Informatics Association*, 27, 12 2019. 4, 15

[27] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *Computing Research Repository*, abs/1803.01271, 2018. 4, 9, 10, 15, 20, 21

[28] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015. 4, 8, 15

[29] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. 4, 7, 15

[30] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural Computation*, 31(7):1235–1270, 2019. 4, 8, 15

[31] Siu Lau Ho and Min Xie. The use of arima models for reliability forecasting and analysis. *Computers & industrial engineering*, 35(1-2):213–216, 1998. 6

[32] Kenneth Gilbert. An arima supply chain model. *Management Science*, 51(2):305–310, 2005. 6

[33] Adebiyi A Ariyo, Adewumi O Adewumi, and Charles K Ayo. Stock price prediction using the arima model. In *2014 UKSim-AMSS 16th international conference on computer modelling and simulation*, pages 106–112. IEEE, 2014. 6

[34] Nesreen K. Ahmed, Amir F. Atiya, Neamat El Gayar, and Hisham El-Shishiny. An empirical comparison of machine learning models for time series forecasting. *Econometric Reviews*, 29(5-6):594–621, 2010. 6

[35] Yao Qin, Dongjin Song, Haifeng Chen, Wei Cheng, Guofei Jiang, and Garrison Cottrell. A dual-stage attention-based recurrent neural network for time series prediction. *arXiv preprint arXiv:1704.02971*, 2017. 8, 20

[36] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014. 8, 9, 20

[37] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. 8

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. 8, 20

[39] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. 9, 20

[40] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. 10, 11

[41] Tcrts terminology and notation. `https://site.ieee.org/tcrts/education/terminology-and-notation/`. Accessed: 2021-09-16. 10

[42] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009. 12

[43] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *expert systems with applications*, 36(10):11994–12000, 2009. 12

[44] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013. 12

[45] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016. 12

[46] Omar Y. Al-Jarrah, Carsten Maple, Mehrdad Dianati, David Oxtoby, and Alex Mouzakitis. Intrusion detection systems for intra-vehicle networks: A review. *IEEE Access*, 7:21266–21289, 2019. 12

[47] Wufei Wu, Renfa Li, Guoqi Xie, Ji yao An, Y. Bai, Jia Zhou, and Keqin Li. A survey of intrusion detection for in-vehicle networks. *IEEE Transactions on Intelligent Transportation Systems*, 21:919–933, 2020. 12

[48] Mirco Marchetti and Dario Stabili. Anomaly detection of can bus messages through analysis of id sequences. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1577–1583, 2017. 12, 13

[49] Markus Hanselmann, Thilo Strauss, Katharina Dormann, and Holger Ulmer. Canet: An unsupervised intrusion detection system for high dimensional can bus data. *IEEE Access*, 8:58194–58205, 2020. 13, 17

[50] Yijie Xun, Jiajia Liu, Nei Kato, Yongqiang Fang, and Yanning Zhang. Automobile driver fingerprinting: A new machine learning based authentication scheme. *IEEE Transactions on Industrial Informatics*, 16(2):1417–1426, 2020. 14, 17

[51] David M.J Tax and Robert P.W Duin. Support vector domain description. *Pattern Recognition Letters*, 20(11):1191–1199, 1999. 14

[52] Omer Berat Sezer, Mehmet Ugur Gudelek, and Ahmet Murat Ozbayoglu. Financial time series forecasting with deep learning : A systematic literature review: 2005–2019. *Applied Soft Computing*, 90:106181, 2020. 14

[53] Liang Zhao. Event prediction in the big data era: A systematic survey. *ACM Comput. Surv.*, 54(5), may 2021. 16

[54] Guillaume Dupont, Alexios Lekidis, J. (Jerry) den Hartog, and S. (Sandro) Etalle. Automotive Controller Area Network (CAN) Bus Intrusion Dataset v2, `https://data.4tu.nl/articles/dataset/Automotive_Controller_Area_Network_CAN_Bus_Intrusion_Dataset/12696950`. *4TU.Centre for Research Data*, 11 2019. 18

[55] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2014. 18

[56] Dirk Ziegenbein Simon Kramer and Arne Hamann. Real world automotive benchmarks for free. *In International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015. 18

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 20

[58] Colin Lea, Michael D. Flynn, René Vidal, Austin Reiter, and Gregory D. Hager. Temporal convolutional networks for action segmentation and detection. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, pages 1003–1012. Institute of Electrical and Electronics Engineers Inc., November 2017. 20

[59] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017. 32