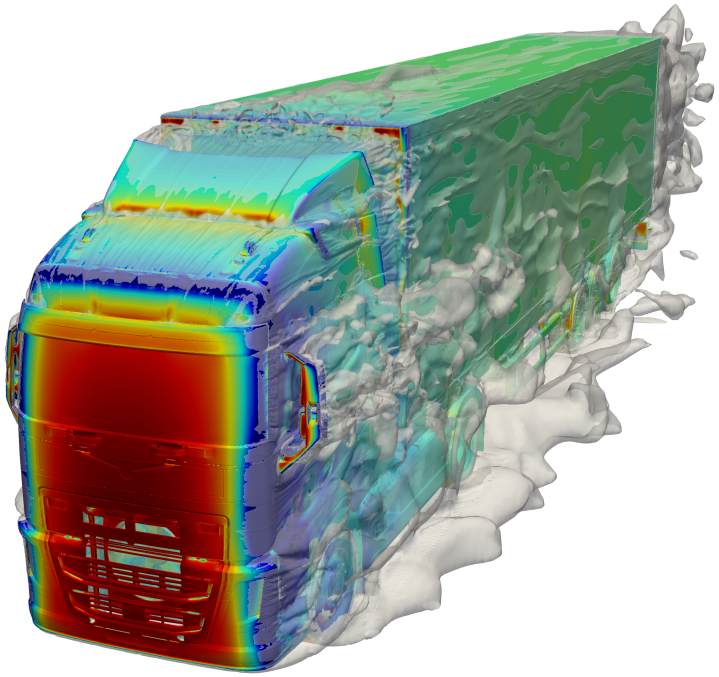
CHALMERS
UNIVERSITY OF TECHNOLOGY



Using neural networks to predict the optimal roof deflector position on trucks

Master's thesis in Systems, Control and Mechatronics

ANTON FAHLGREN

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019:NN

Using neural networks to predict the optimal roof deflector position on trucks

ANTON FAHLGREN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2019

Using neural networks to predict the optimal roof deflector position on trucks
ANTON FAHLGREN

Supervisor: Andreas Persson, Rumblestrip
Examiner: Martin Fabian, Department of Electrical Engineering

Master's Thesis 2019:NN
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Truck cab and trailer with a correctly positioned roof air deflector.

Typeset in LaTeX
Gothenburg, Sweden 2019

Using neural networks to predict the optimal roof deflector position on trucks
Anton Fahlgren
Department of Electrical Engineering
Chalmers University of Technology

Abstract

In an attempt to meet the increasing demands on the transport sector, the heavy truck industry is focused on finding alternative ways beyond the scope of an efficient powertrain to reduce its environmental impact. A roof air deflector is an aerodynamic component mounted on the truck cab's roof with the purpose of reducing the total air resistance of the truck. The deflector has to be adjusted manually whenever the trailer is changed in order to minimize the achieved fuel consumption. Unfortunately, this process is often overlooked in practise. As a result, efforts have been made to automate the adjustment and finding of optimal roof deflector position to remove the necessity of driver interaction.

In this work, an approach for predicting the optimal roof deflector position of a truck using convolutional neural networks is implemented and evaluated. The prediction algorithm uses current consumption data from a linear actuator that moves the roof deflector as its only input. The current consumption has been sampled during a sweeping motion of the roof deflector while the truck is driving with a range of different trailers. This makes it possible to detect the aerodynamic at different roof deflector positions through the current samples

Results show that a neural network based approach is feasible and the algorithm is capable of consistently predicting the optimal position for previously unseen data. The proposed algorithm is completely end-to-end and uses convolutional layers for extracting discriminating features in order to predict the correct position.

Keywords: active aerodynamics, deep machine learning, signal processing, time series classification, artificial neural networks, convolutional neural networks

Acknowledgements

I would like to start by expressing my gratitude to the company Rumblestrip. You made this thesis possible with your warm welcome to the team and unconditional sharing of knowledge. Moreover, seeing everyone's dedication and excitement about the product and the company as a whole has been very inspiring.

I would also like to specifically acknowledge my supervisor at Rumblestrip, Andreas Persson, who provided me with invaluable feedback and answered any questions I had throughout this project.

Futhermore, I would like to thank Martin Fabian, my examiner at Chalmers University of Technology, for taking on this thesis. Never during the project have I felt uncertain of the expectations on me and I have your clear directions and constructive feedback to thank for that.

Finally, to all my nearest and dearest who have supported me through my years of studies - Thank you.

Anton Fahlgren, Gothenburg, May 2019

Contents

Acronyms

CNN	Convolutional Neural Network
FCNN	Fully Convolutional Neural Network
FCP	Fuel Consumption Penalty
GADF	Gramian Angular Difference Field
GAF	Gramian Angular Field
GASF	Gramian Angular Summation Field
IM-CNN	Image based Convolutional Neural Network
LSTM	Long Short-Term Memory
NOP	Number Of Parameters
RAD	Roof Air Deflector
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
RP	Recurrence Plot
SE	Squeeze-and-Excitation
TSC	Time Series Classification

Nomenclature

d	Dropout rate
k	Complexity scaling

1
Introduction

1.1 Background

The transport sector, and in particular the heavy truck industry, is under constant scrutiny from customers and authorities in regards to its environmental impact. New regulations for air pollution and carbon emissions have been passed continuously throughout recent years. In an attempt to meet the increasing demands, the heavy truck industry is focused on finding alternative ways, beyond the scope of an efficient powertrain, to reduce the environmental impact.

Roof air deflectors (RAD) are commonly used aerodynamic components on trucks to reduce the air resistance, which in turn reduces fuel consumption. Typically, the deflector's position needs to be manually adjusted whenever the truck changes to a trailer with a different height or gap to the truck cab. This adjustment is often overlooked as it requires taking several measurements between the cab and trailer and then finding the optimal position in the truck's handbook. The result is that many trucks go long periods of time with an incorrectly positioned RAD, leading to unnecessarily high fuel consumption.

The company Rumblestrip has developed an adaptive RAD, mounted on linear electric actuators, in order to automate the positioning procedure. After the truck has changed trailer and reached a certain speed, the adaptive RAD does a sweeping motion while measuring current consumption and extension state of the actuators. Because the current consumption contains information of the aerodynamic forces acting on the RAD, its time series can be used to predict the optimal position. This is achieved by training a model using labeled data and supervised learning methods.

1.2 Related work

The problem of time series classification (TSC) has been heavily researched in recent years. Traditionally, state-of-the-art TSC algorithms have largely been based on nearest neighbor (NN) classifiers using alternative distance metrics such as dynamic time warping [1]. However, the main drawback of NN classifiers is the computational cost and storage requirements since the entire training set has to be searched for every prediction. Less computationally heavy approaches have been shown to yield good performance, but often rely on hand engineered features extracted from the raw time series [2, 3]. Furthermore, useful features are often domain specific and can be difficult to find.

With the advancement of machine learning and deep neural networks, several

methods to derive end-to-end TSC algorithms have been proposed. This eliminates the need for manual feature engineering and the models can be trained on mostly raw data. The simplest neural network type is the multi layer perceptron, also known as fully connected. This type of network is not well suited for TSC as the structure does not take temporal information into account. Instead, the time series samples are treated independently, which can be detrimental to the accuracy of the classifier [4].

Convolutional neural networks (CNN) are typically used for image classification because of their ability to capture spatial structure in the data by sliding filter over the data. The same methodology can be applied to time series by sliding the filter across the temporal dimension rather than spatial. Several studies show that CNNs, alone or in conjunction with other network types, are able to accurately classify time series in multiple areas of application. In [5], a pure end-to-end CNN model is proposed and shown to achieve comparable performance to other state-of-the-art methods. Furthermore, the network structure used makes it possible to compute a class activation maps, which illustrates which sections in the data that contributes to specific labels. The CNN architecture from [5] is further developed in [6] by augmenting it with a recurrent neural network (RNN) module called long short term memory (LSTM). The LSTM cell processes a dimension shuffled version of the input data, and its output is concatenated with the CNN's output before the final layer. The proposed methodology is shown to increase accuracy with a nominal increase in number of parameters.

A review and benchmark of nine end-to-end deep learning architectures for TSC is conducted in [4], using both uni- and multivariate data sets. It is shown that for small data sets some CNNs tend to overfit the training data. However, algorithms that deploy data augmentation steps before training tend to overfit significantly less. A technique for augmenting time series data called window slicing is presented in [7]. Here, each time series is sliced into several smaller segments and then treated as separate training instances. At test time, the same slicing is conducted and the prediction is computed using the majority vote. It is shown that this augmentation can reduce the overfitting problem for CNNs.

Inspired by computer vision research, several techniques to transform time series data into images before classification have been proposed. In [8, 9, 10], the time series are converted into both Gramian angular fields and Markov transition fields before being fed into CNNs. Converting the time series into recurrence plots before classification has been done in [11, 12]. Both methodologies show good results in terms of classification accuracy. These image transformations are domain agnostic and enables the use of well established image classification methodologies [4].

RNNs, and in particular LSTM, have been shown to have robust performance in natural language processing tasks [13, 14]. While time series have a lot of similarities to the sequenced nature of sentences, pure RNNs are rarely applied to TSC problems because they are considered difficult to train and parallelize [4]. In [15], the speed and convergence of the training is improved by training the RNN model for time series forecasting and classification at the same time. However, the resulting classification accuracy is no better than typical CNN models across several data sets. Nonetheless, in [16] it has been demonstrated that a pre-trained RNN can

be used to extract features from raw time series across several application domains with good performance. The extracted features can then be fed to any classifier.

1.3 Purpose

The purpose of the project is to aid in the development of an algorithm that can predict the optimal roof deflector position based on sensor data from the actuators and truck. Various neural network based approaches are researched and evaluated in regards to their predictive performance and computational complexity. Results should indicate if prediction using neural networks is feasible for use in the embedded application and if the computational hardware can be a bottleneck for the predictive performance.

1.4 Scope

The thesis focuses on the comparison and evaluation of different deep neural networks trained to predict the optimal RAD position. The work targets low complexity neural network structures that are feasible for deployment on a resource constrained microcontroller. Primarily, the current consumption of the actuators and corresponding correct roof deflector position is used as training and test data for the models.

1.5 Objective

The goal of the thesis is to provide a comparative study and implementation of several deep neural network based prediction methods. In order to achieve this goal, there are various tasks and problems that need to be addressed:

- Literature study about using deep neural networks for TSC
  - What type of network architectures are typically used?
  - What are their main benefits and drawbacks?
  - Which architectures are suitable for running on resource constrained hardware?
- Based on the literature study, determine which network architectures to proceed with
- Establish a framework to work with deep machine learning tools
  - Determine suitable software environment to use
  - Ensure compatibility with the data set and comparison with existing algorithms
  - Set up cloud computing capabilities for efficient training and evaluation
- Implement and train the deep neural network models
  - What type of loss function should be used during training?
- Evaluate the trained models
  - Establish performance metrics to be used for evaluation
  - Analyze the predictive performance on unseen data

- Compare the computational complexity of the trained neural network models
- Evaluate how the performance is influenced by various data set characteristics

2
Theory

2.1 Supervised learning

Supervised learning is a class of machine learning problems where input and output pairs are used to teach a machine to identify the relation between the variables. I.e. for a given set of collected data points

$$(x, y),\ x \in X,\ y \in Y \tag{2.1}$$

the objective of supervised learning is to find a function f such that

$$f : X \to Y \tag{2.2}$$

by training a model using example pairs of input and output data. The trained model can then be used for inference to generate a prediction, $\hat{y}$, for previously unseen input samples.

Typically, the choice of algorithm to accomplish this varies a lot in different areas of application. However, there are some general factors to consider when dealing with supervised learning tasks.

2.1.1 Bias-variance

For a trained model, the expected prediction error, often called generalization error, can be divided into several components using bias-variance decomposition [17, 18]. For a given input sample x and true function $f(x)$, the expected prediction error of the estimated model $\hat{f}(x)$ becomes

$$\mathbb{E}[(f - \hat{f}(x))^2] = (Bias[\hat{f}(x)])^2 + Var[\hat{f}(x)] + \sigma_e^2 \tag{2.3}$$

Here, the error due to bias is defined as the difference between the expected prediction of the model and the true output, according to

$$Bias[\hat{f}(x)] = f(x) - \mathbb{E}[\hat{f}(x)] \tag{2.4}$$

and the error due to variance is a measure of how much the prediction for a given data sample x varies. Alternatively, it can be seen as the variance of the used learning method. It is mathematically defined as

$$Var[\hat{f}(x)] = \mathbb{E}[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2] \tag{2.5}$$

Finally, σ_e^2 is the irreducible error due to the noise in the data set used. Therefore, when developing a model through supervised learning, the ultimate goal is to minimize the bias and variance error. However, how much these two error sources can be reduced is limited by the so called bias-variance tradeoff.

Typically, a model with a high bias is too simplistic to capture the characteristics of the underlying process. This is often called underfitting and is the consequence of using a model structure that is too inflexible. To reduce bias, a more complex and flexible model can be used. However, doing this increases the variance due to the tradeoff property. A model with high variance may have good performance on the training set, but likely has very poor performance on a test set. This is because the high flexibility leads to noise in the data being modelled while training. When evaluating the model on the test set, the noise component is entirely different, and the error rates become higher as a result. This phenomenon is known as overfitting.

For a given data set, the optimal fit is the one that minimizes the total error. In practise, it is impossible to accurately decompose the error contribution due to the stochastic nature of the noise component. As a result, finding a model that strikes a good balance between bias and variance is one of the most difficult parts of any supervised learning problem.

2.1.2 Data set characteristics

The characteristics of the data set, and by extension the underlying process, play a significant role when deciding on what algorithm to use. In general, if the true function that the data is drawn from is simple, less data is required to learn a good approximation. Furthermore, this can often be accomplished using a relatively inflexible learning algorithm. When the true function is more complex, a more flexible learning algorithm and significantly more data is often needed to reach a good approximation.

Another factor is the dimensions of the input data. If the input space dimension is large and contains many features that are not relevant for mapping input to output, some learning algorithms may struggle to find a good approximation. This is because a lot of the flexibility of the algorithm may be wasted on modelling irrelevant input features. In such cases, performance can often be increased by removing unneeded features or choosing an algorithm with lower variance and flexibility. There also exists learning algorithms that attempts to automatically determine the importance of different features in order to avoid making assumptions about the underlying function and data. Furthermore, strongly correlated or redundant features can make certain learning algorithms struggle to find a good approximation of the true function.

The magnitude and type of noise present can also have an effect on what learning algorithm is appropriate for the data set. If the output data contains noise, the algorithm should not attempt to perfectly fit the output values. Doing so would mean the learned function uses some expressiveness to model noise. This could result in large generalization errors when the algorithm is used for inference on unseen data.

In addition to the internal characteristics of the data set, the actual size of the

data set can have a large impact on the outcome of the applied machine learning approach. With more observation available it often becomes easier for models to find the underlying function as it is given many more examples of this but with varying noise components. Nonetheless, it is not a universal truth that more training data yields better performance. If the chosen algorithm is of very low complexity it may have already plateaued in terms of its performance. If there are no degrees of freedom left in the model, then more data will not necessarily yield any returns. Determining if this type of plateauing is occurring or not can be of high importance if it possible to gather more training data. In order to do this, one can deliberately withhold a portion of the training data and train the model on a small subset rather than the full data set. By doing this several times, with a different ratio of the training data at each step, one can determine if the size of the training set influences the performance.

2.1.2.1 Splitting data

As discussed, the performance of a machine learning algorithm is largely affected by the characteristics and size of the data set. Another important aspect is how that data set is actually used. While one may want to use all of the available data set for training, doing so eliminates the possibility of objectively evaluating the performance of the trained model. Therefore, to be able to compare and evaluate various models a portion of the data set has to be set aside and not used for training as this would yield unrealistically optimistic performance results. If the set used for testing the trained model has no overlap with the set used for training, the performance on the test set is a much more reliable measure of how good the model is.

In cases where the machine learning approach has a lot of tunable hyperparameters, a validation set may also be used. The idea is then that the training set is used to learn the actual model parameters. The validation set is then used to tune hyperparameters such as network architecture, model complexity or optimization settings. Furthermore, it is used to estimate the performance of the evaluated models in order to select the best one. Finally, the test set is used to get an unbiased estimate of the algorithms performance as none of those data observations have been used to train the model or select hyperparameters. As a result, it is not always strictly necessary to use separate validation and test sets if a comparison of different models trained and tuned on the same data is the goal. This is particularly true for application where there is a very limited amount of data and it is desirable to use as much as possible for training. However, in the era of big data, it is very common to use all three splits of data as there is often an abundance of observations available. Unfortunately, there are no well established ratios to use for splitting the data into these sets. For example, in [18] it is suggested to use 50% for training and the remainder for validation and testing. However, in [19] it is stated that a common practise is to use 70% for training. Moreover, the author suggests to only use as much validation and test data as is necessary to accurately evaluate the performance of your algorithms.

Another factor to consider is the stochasticity introduced when actually dividing the data into subsets. Two different splits, even though the same number of observation are drawn from the same data pool, may yield very different results. It is

therefore imperative to not only use the same data set, but also the same data split for training and comparing machine learning models. It can also be beneficial to train and evaluate models on several different splits of data. By doing this, it can be determined if the performance results seen are consistent and not just the results of a particularly favorable data split.

2.2 Artificial neural networks

An artificial neural network is a type of mathematical model which structure is inspired by the human brain. The approach revolves around not making any prior assumptions on the nature of the underlying process that one is trying to model. Instead, the learning is exclusively driven by what characteristics the network can identify in the used training data. This is accomplished by building a network using layers of interconnected neurons, or nodes. The output of each neuron is connected to at least one subsequent neuron or is the output of the network. The input of each neuron comes from one or several previous neurons or from the actual input data. The layers between input and output, often denoted as hidden layers, are what defines different types of neural networks. Infinitely many different network structures can be derived by changing the number of layers, neurons, connections and characteristics of each neuron. When using a neural network based approach to supervised learning, determining the best network structure for the task at hand is often the most time consuming and difficult part of the problem. An example of a very simple neural network with two inputs and one output can be seen in figure 2.1 below.

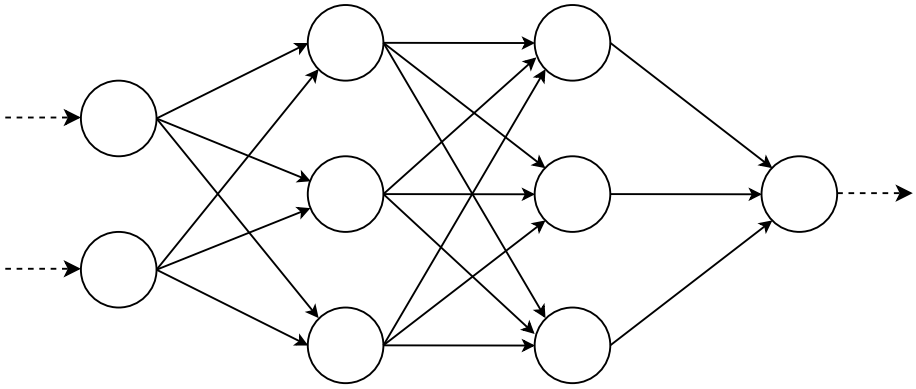


Figure 2.1: Simple fully connected neural network structure with two inputs and one output.

As can be seen, the network has two hidden layers with three neurons in each layer. These hidden layers are where the function approximation takes place. In training, the network attempts to learn the mapping from input to output by looking at a number of examples from a set of input-output pairs. The learning is largely dictated by the characteristics of each neuron and the network structure, which are both determined by the designer in beforehand. Neurons typically perform some mathematical operation, that can be expressed by one or several numerical

parameters, on the input signal. After the mathematical operation, the resulting value is passed through a nonlinear function, called activation function, before being output to subsequent neurons. Common neuron operations are the linear matrix equation and convolutions, but several other operations are possible. The limiting aspect is that the output of the neuron needs to be differentiable with respect to its parameters in order for the learning algorithm to work. This property is what enables the whole network to learn anything from the data. First, the derivative of the output with respect to each of the networks parameters is computed. Then, an optimization algorithm is used to reduce the error between the networks output and training data. This is done by optimizing a loss function using stochastic gradient descent. The optimization passes over the entire training set several times and can run for an arbitrary amount of time. Introducing this stochasticity into the learning algorithm is crucial for the learnability of neural networks. However, this also makes them differ from a lot of traditional machine learning techniques where the learning algorithm typically is deterministic.

Another defining property for neural networks is that the flexibility and expressiveness is infinite in theory. It has been shown by the universal approximation theorem that neural networks can approximate any continuous convex function of n-dimensional input variables. The theorem was first proven by [20], but then only for specific activation functions. Later, [21] improved the theorem by showing that the potential expressive power of neural networks comes from the layered structure rather than the choice of activation function. It was shown that a single layer neural network can be a universal approximator in the single output case if the width of the layer is unbounded. In recent years, [22] and [23] has proven that the universal approximation theorem holds for neural networks that have limited width, but more than one layer. However, the universal approximation theorem does not give any indication as to how many neurons are necessary, the network topology, or the feasibility of learning the correct parameter values.

2.2.1 Fully connected layers

Fully connected layers are very commonly used building blocks in neural networks. They are created by connecting all the outputs of a neuron to all the neurons in the next layer. An illustration of these connections can be seen in figure 2.1 above. With a fully connected structure, the input x of the jth neuron in the ith layer is passed through the linear matrix equation according to

$$a_{i,j} = w_{i,j}x + b_{i,j} \tag{2.6}$$

where $w_{i,j}$ and $b_{i,j}$ is the neurons weight and bias matrices respectively. These matrices contain all the trainable parameters of a fully connected layer. Furthermore, $a_{i,j}$ is the activation of the neuron. This activation is in turned passed through the neurons activation function f to compute the output $z_{i,j}$ as

$$z_{i,j} = f(a_{i,j}) \tag{2.7}$$

These operations can be summarized into a single equation for computing the output of all the neurons in the ith layer according to

$$z_i = f(W_i z_{i-1} + B_i) \tag{2.8}$$

where z_{i-1} is the output from the previous layer. Because of this matrix multiplication, the number of trainable parameters is directly proportional to the number of neurons in the layer and the input dimension.

2.2.2 Convolutional layers

Convolutional layers are in a lot of ways similar to fully connected layers. They consist of neurons with weights and biases that can be trained in a similar fashion. However, instead of the matrix multiplication seen in fully connected layers, convolutional layers uses the convolution operator to compute the activation of the neuron. Here, the input signal is convolved with a filter, with dimensions manually chosen, before being passed to the activation function. In order to reduce the number of parameters in a convolutional layers, the filter coefficients are shared across the dimension that the convolution is computed along. This works particularly well for time series and image data, where the convolution is computed across time and spatial dimensions respectively. By doing this, the number of trainable parameters is significantly reduced. Furthermore, this enables convolutional layers to be used on data with none or minimal pre-processing. Extracting useful features from the large dimension input then becomes part of the learning task. The actual filter coefficients that are needed for this are learned automatically in a similarly to fully connected layers.

The characteristics of a convolutional layer is defined by several design parameters. First, the number of neurons in the layer has to be determined. Secondly, the filter dimensions, also known as receptive field, have to be defined. Lastly, parameters for the convolution itself needs to be set. The stride parameter denotes how much the receptive field moves for every step in the convolution. An output is computed at every step, so by using a large stride the output dimension can be reduced. Zero padding is another parameter for the convolution. This can be used to control the dimension of the output, by concatenating the input signal with zeros before computing the convolution. It is typically used to keep the input and output dimensions equal. After convolution operation, the signal is passed through an activation just as in the fully connected layers.

While it is possible to define convolutional layers over any dimension, with and without parameter sharing, the most commonly used convolutional layers are

- **Temporal convolutional layers**, which are used with time series data x_{ts} and a one dimensional filter h_{ts}. The output y_{ts} of the convolution is then a time series and is computed along the time axis according to

$$y_{ts}[i] = \sum_{k=-\infty}^{\infty} x_{ts}[k] h_{ts}[i-k] \tag{2.9}$$

- **Spatial convolutional layers**, which are used with image data x_{im} and a two dimensional filter h_{im}. The output y_{im} of the convolution is then an image and is computed along the spatial axes according to

$$y_{im}[i,j] = \sum_{k_i=-\infty}^{\infty} \sum_{k_j=-\infty}^{\infty} x_{im}[k_i,k_j] h_{im}[i-k_i, j-k_j] \tag{2.10}$$

In both of these cases, the convolutional layers automatically learn to extract useful features from the time series or images in order to perform regression or classification. Furthermore, this type of structure can be used with data that has more than one channel, for example xyz readings from an accelerometer or images represented with an RGB color model.

2.2.3 Recurrent layers

Recurrent layers are different from normal feed forward layers because they keep an internal state, also known as memory. This enables the layers to have temporal dynamic behaviour, which has been shown to very useful when dealing with sequential data. Recurrent layers are typically used with both input and output data being sequences, but it is possible to use them without sequential altogether. The result is that while processing a sequence, a recurrent layer does not start from scratch on every sample when it computes the activation. Consider an input sample at a given time x. A recurrent layer computes the output y using both the input and an internal state h. This internal state is also updated when the output is calculated. At the next sample, the output is calculated in exactly the same way, but now the internal state of the layer is different. An illustration of how a recurrent layer can be seen in figure 2.2 below. It shown in both a condensed and unrolled form.

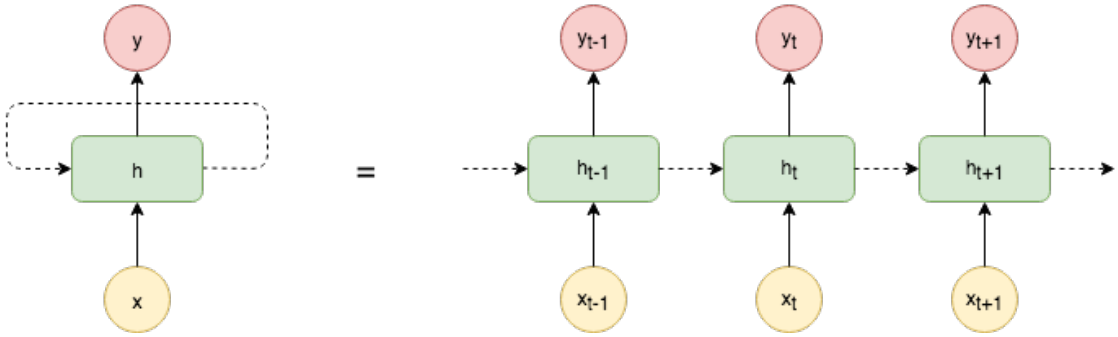


Figure 2.2: Simple recurrent layer in both its condensed and unrolled form.

Now consider a recurrent layer with fixed parameter values. The computed output for a given input sample's value is different depending on the characteristics of the previous input samples. This is very different from a traditional feed forward layer, where a given input value always maps to the same output value, regardless of how the input looks before and after in time. This temporal dynamic behaviour has been shown to be very effective when dealing with for example machine translation problems. Recurrent layers then enable the model to translate words while taking the context of the sentence into account, rather than simply translating one word at a time independently.

One major drawback of recurrent layers is that they are notoriously difficult to train [24]. In order to update the network parameters using optimization all the gradients in the network have to computed. To do this for a recurrent layer it has to be unrolled, and gradients have to be computed backwards in time. This

involves computing gradients for every instance of the layer, as it changes over time while processing a sequence. Unrolled recurrent layers tend to be very deep, which leads to something called the unstable gradient problem. This problem can yield gradients that tend toward zero or infinity, which stops the optimization process and by extension the learning.

2.2.3.1 Long short-term memory layers

The LSTM layer is a variation of the traditional recurrent layer. The major difference is the addition of a forget gate, which allows the layer to selectively choose what should be kept in the layer's internal state. An illustration of the LSTM layer architecture can be seen in figure 2.3 below. Here, σ is the sigmoid function and *tanh* is the hyperbolic tangent function.

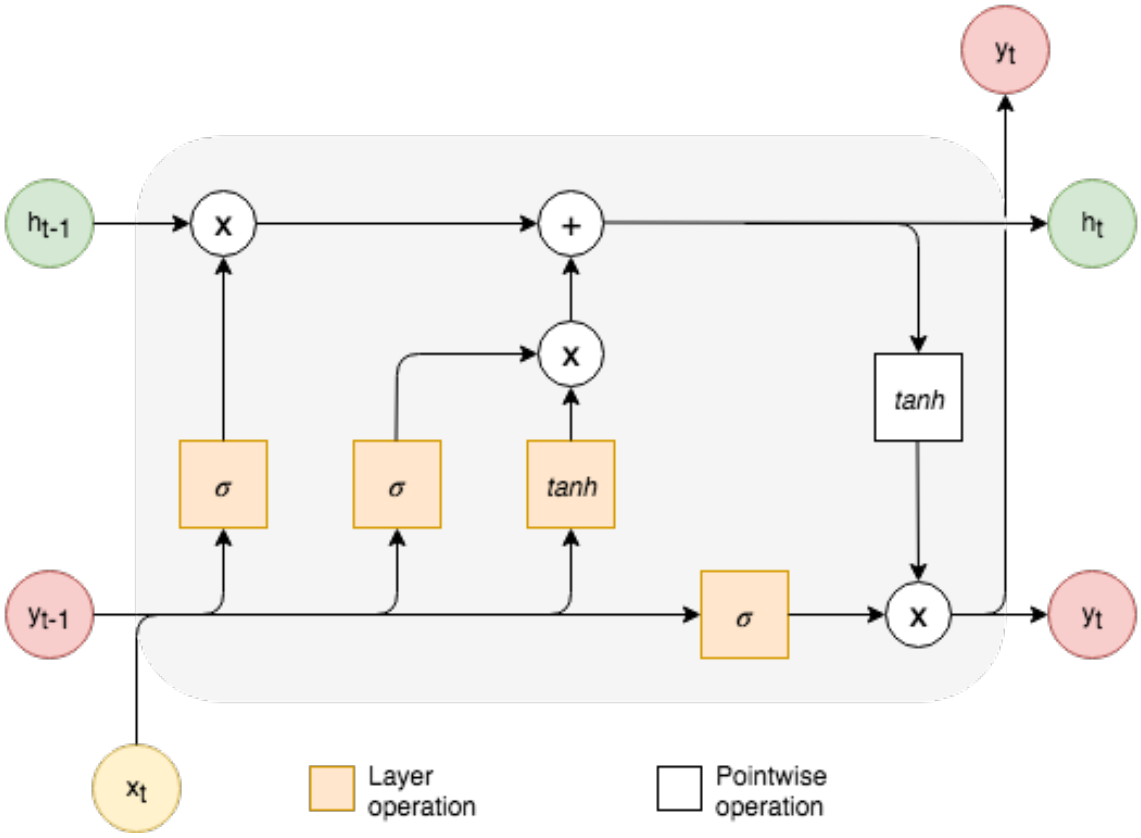


Figure 2.3: Architecture of a long short-term memory layer.

This layer architecture is significantly more complex than the normal recurrent layer seen in figure 2.2. However, the augmentations allow LSTM layers to model long term dependencies in sequential data without exhibiting the unstable gradient problem as in regular recurrent layers. Due to this, networks built with LSTM layers instead of normal recurrent layers have much better convergence rate during training with very little drawbacks. As a result of these improvements, the LSTM layers are commonly used in practise and have been used to break several previous records in areas such as machine translation, language modeling and image captioning [25, 26, 27].

2.2.4 Pooling layers

Pooling layers are commonly used together with convolutional layers because of their ability to reduce the dimensionality of the input. They contain no trainable parameters, but rather just carry out a simple pooling operation on the layers input features. By pooling the features as they propagate through the network, much faster training and prediction computations can be achieved. It can also serve as a way to reduce overfitting in the model.

The pooling operation can be carried out along any dimension of the input, but it is typically used along the temporal or spatial axes. The operation consists of moving a window across the chosen dimension and at every step it summarizes the content of the window into a single value. Typically, this is done by either averaging the value in the window, or by extracting the maximum value in the window. The dimensions of the pooling layers output is therefore a function of the window size and stride. An illustration of the average and maximum pooling operations applied on a temporal input can be seen in figure 2.4 and on an image input in figure 2.5.

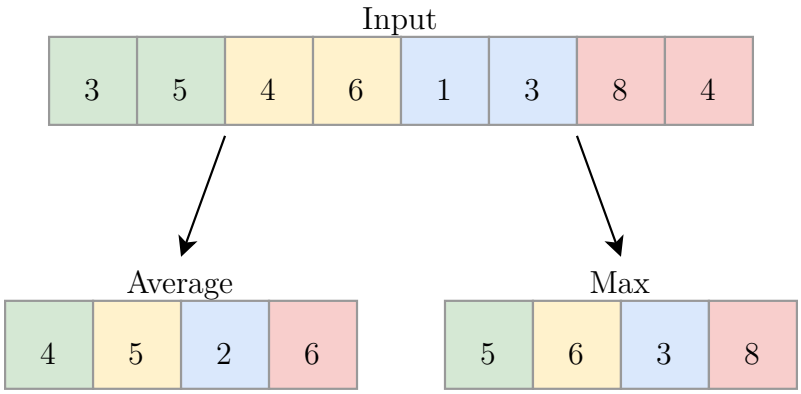


Figure 2.4: Example of the average and max pooling operations carried out on one dimensional data.

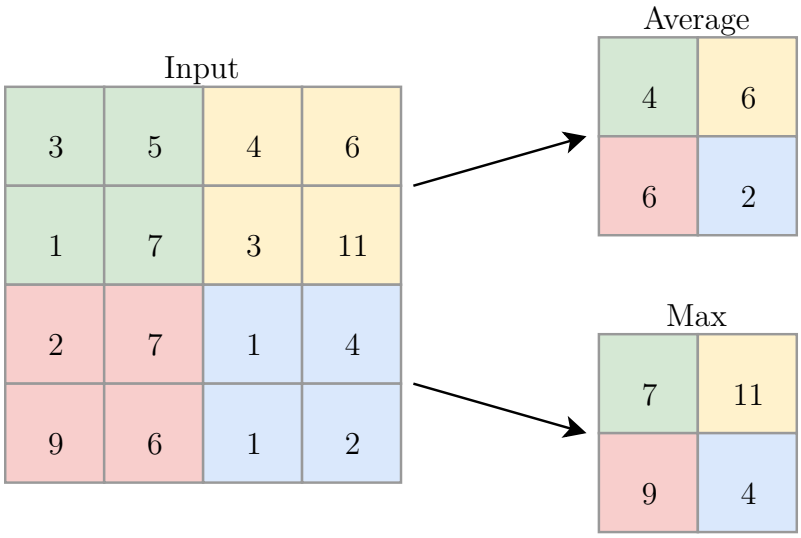


Figure 2.5: Example of the average and max pooling operations carried out on two dimensional data.

In both examples the pooling layers use a windows size of 2 and a stride of 2. This reduces the feature size to half for both the temporal and image data. The special case when the window size is equal to the input dimension is often denoted as global pooling. A global pooling operation summarizes the input into a single values along a given dimension.

2.2.5 Activation functions

As previously mentioned, activation functions play a crucial role in computing the output of neurons in a network. Depending on the area of application and where in the network they are located, different activation functions are used. However, the common denominator among typical activation functions is that they are all nonlinear. Due to the layered structure of neural networks, any network with only linear activation functions could be replaced with a single layer and achieve the same input to output mapping. This is because any linear combination of linear functions is just another linear function. Having nonlinear activation functions is therefore essential to the success of deep neural networks. Furthermore, the universal approximation theorem has only been shown to for neural networks with nonlinear activation functions.

Typical choices of activation functions are step functions, sigmoids, hyperbolic tangent (TanH) and variations of the rectified linear unit (ReLU). In some cases, intuitions about the data characteristics and underlying process can help in choosing what activation functions to use in the network. Nevertheless, it is often a trial-and-error procedure. In figure 2.6, a few examples of commonly used activation functions can be seen.

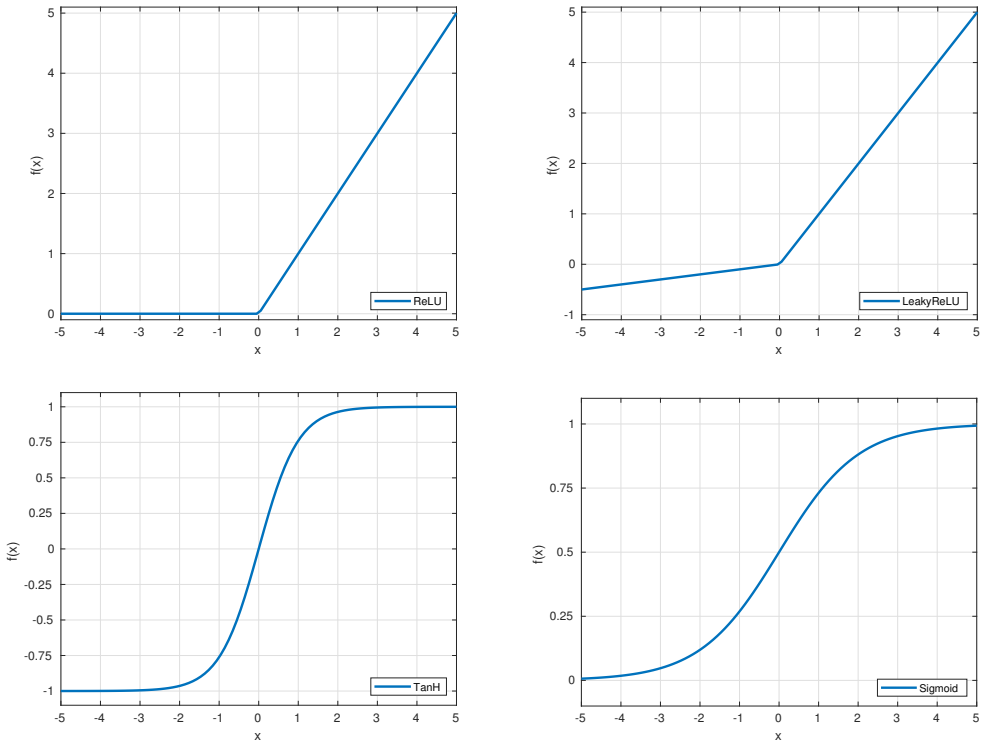


Figure 2.6: Four commonly used activation functions in neural networks.

For classification problems, the desired output is often the probability that a sample belongs to a certain class. For this purpose, a sigmoid function is often used in the output layer as it yields values in the range $[0, 1]$. For multiple class classification problems a variation of the sigmoid function, called softmax, is often used. This activation function normalizes the output probabilities of the sigmoid function such that the sum of probabilities is 1. For regression, the choice of activation function in the output layer is typically identity mapping so that the output may take any values.

For hidden layers, ReLU is often the go to option because of its simple gradient. During training, activation functions like TanH and sigmoid can lead to problems with vanishing gradients due to the low rate of change in the functions during certain intervals. This in turn leads to weights not being updated, which stop the learning process. The two are also computationally heavier than ReLU, which can be of importance when designing very deep neural networks. The ReLU activation function does exhibit some issues with the gradients as well. The horizontal component of the activation can lead to some neurons being unresponsive during training because of the gradient being zero. Variations of the ReLU where the negative side has a slope can be used to remedy this. The LeakyReLU function seen in figure 2.6 is one example of this. Even if ReLU has some potential issues with the gradients, empirical results show that the convergence speed is often several times faster for networks with ReLU activation in the hidden layers than for networks using sigmoid or TanH activations.

2.2.6 Batch normalization

The use of very deep and complex neural networks has become increasingly popular in recent years, as the collection of very large sets of training has become feasible. Using more complex models often results in much slower training times and converge speeds. In some cases, a too complex model may not converge at all. As flexible neural networks are often desired because of the complex true function, several techniques have been proposed to improve convergence and training speeds. One common approach is to normalize the training data to zero mean and unit variance. This ensures that the training does not start in an area where the derivatives of the activation functions are close to zero, which improves the convergence speed [28, 29]. Batch normalization is another technique that takes this one step further by normalizing, scaling and shifting the input of every layer in the network. The batch normalization operation is typically computed before the activation function of a neuron. More specifically, given a batch of training data x with m number of features, the batch mean μ_B and variance σ_B^2 is calculated for each feature as

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.11}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.12}$$

Then, using the batch mean and variance, the input of the layer is normalized to zero mean and unit variance according to

$$\bar{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.13}$$

where ϵ is small constant added for numerical stability. Simply normalizing each layers input could change what the layer can represent. For example, in the sigmoid case the normalization constraints the inputs to the linear section of the nonlinear activation functions. Therefore, a scaling and shift operations is done to ensure that the batch normalization can yield an identity transformation. For the normalized input $\bar{x_i}$, the output y_i of the batch normalization is then computed as

$$y_i = \gamma_i \bar{x_i} + \beta_i \tag{2.14}$$

where γ and β are additional parameters of the network that are learned during training. Note that using $\gamma = \sqrt{\sigma_B}$ and $\beta = \mu_B$ would yield the same output as without the batch normalization layer inserted. Even with the additional parameters in the network, empirical results shown that adding batch normalization layers often significantly reduce training time [30].

Despite the widespread adoption of batch normalization layers in neural networks, there is no well established explanation for why they work so well. It has been proposed that batch normalization helps with convergence by reducing the internal covariate shift in the network [31]. Internal covariate shift is defined as the change in distribution in activations due to training updates of the network parameters. It is suggested that batch normalization whitens these distribution, which in

turn benefits the training. In [30] it is shown that batch normalization might not actually reduce internal covariate shift. Instead, authors suggest that the success of batch normalization is due to a smoothing effect that makes the gradients in the optimization more predictive. This enables the use of larger learning rate and leads to faster convergence rates.

2.2.7 Dropout

Dropout is a stochastic regularization technique that was originally proposed in [32]. It serves as a method to reduce overfitting when training neural networks. The general idea is that at each training pass, one or several nodes in the network are dropped out of the network. In practise, this means removing the nodes from the network, along with all of its connections. The sampled network after dropout is a thinned version of the original neural network, as an arbitrary number of nodes and connections have been excluded. An example of a thinned network can be seen below in figure 2.7, where two nodes have been removed from the fully connected neural network structure.

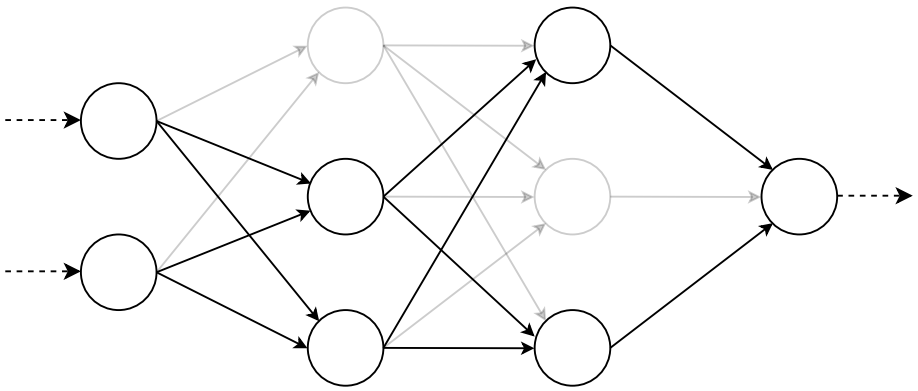


Figure 2.7: Fully connected network where one node in each hidden layer is excluded due to dropout.

Now consider a neural network of n nodes. This network can be expressed as a collection of 2^n unique thinned neural networks. Note that the number of parameters stays the same, as all thinned networks in the collection share parameter values. At every training step, one of these networks are sampled and used to update the trainable parameters as per the normal learning procedure. At the next step, another network is sampled randomly to be used for training. This procedure could potentially generate up to 2^n number of trained networks, which is typically not feasible to use at test time as it would require combining the prediction from every single network in the collection. Therefore, the dropout technique uses a simple approximating averaging method. Given that node has been kept with a probability p during the training stage, the output of that node is then multiplied with p when used for inference. Doing this ensures that the output during test time is equal to the expected output during training time. By applying the approximation method the collection of networks can be condensed into a single trained neural network that can be used for prediction.

One of the drawbacks of using the dropout technique is that the number of passes over the training data, called epochs, required to converge increases. This happens because the parameters in nodes that are excluded are not updated for a given training step. However, while the number of epochs grows, the time to complete an epoch decreases with more dropout. With less active nodes at each step, the computation of derivative during training takes less time, which means that a single pass over the data is faster than without dropout. Another factor to consider is that using dropout reduces the expressive power of a neural network with a given number of nodes, as not all of them are active all the time. In [33] it is suggested that this can be remedied by increasing the number of nodes n in a layer to n/p, if dropout with retaining rate p is applied after the layer. Finally, as dropout introduces a type of noise to the training procedure, some adaptations to the learning rate and optimizer settings may be necessary to reach satisfactory results when dropout layers are used.

2.2.8 Squeeze-and-excitation blocks

Squeeze-and-excitation (SE) blocks were originally proposed in [34] as an augmentation to CNNs. The aim of the SE block is to improve the predictive performance by capturing dependencies between different channels in the network. This is accomplished by adding a block that allows the network to process global, cross channel, information in order to extract useful features. The SE block was initially proposed for input data formatted as images, but the same operations can be applied in any context where convolutional layers are used as they produce outputs that consist of features with multiple channels. In figure 2.8, the SE operation is illustrated for a set of image features.

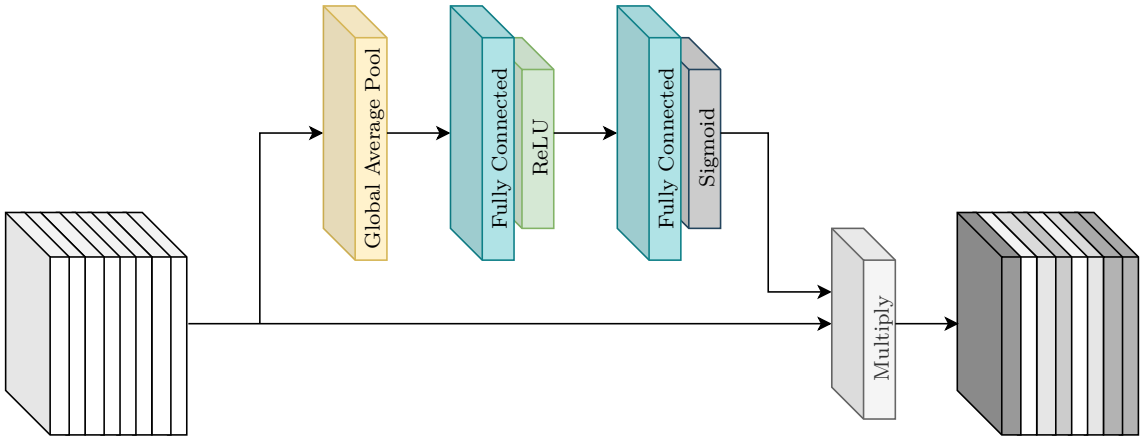


Figure 2.8: Architecture of an SE block.

First, the features are pooled across the height and width dimensions using a global averaging layer. The averaged features are then passed through two fully connected layers. The first one with a ReLU activation function and the second one with a sigmoid activation function. Finally, the output of the fully connected layer is multiplied with the original features to produce the final output of the SE block. In the end, the SE block is simply a scaling on the input features in the channel

dimension. However, this scaling is determined by the channel-wise dependencies that the fully connected layers manage to identify.

2.2.9 Parameter learning

Learning the parameters of a neural network is done iteratively using optimization techniques. The general idea is that the network is presented one or several input-output pairs at a time. Then, the parameter values are updated such that the error between the predicted outputs and the true labels is minimized. When the last sample has been used, the process start over from the first sample again. One such pass over the training data set is typically called epoch and neural networks often need to be trained for many epochs before good convergence is achieved.

Because of the connectionist nature of neural networks, all the nodes in the network have to be taken into account to perform a single parameter update. Consider the network seen in figure 2.9 below.

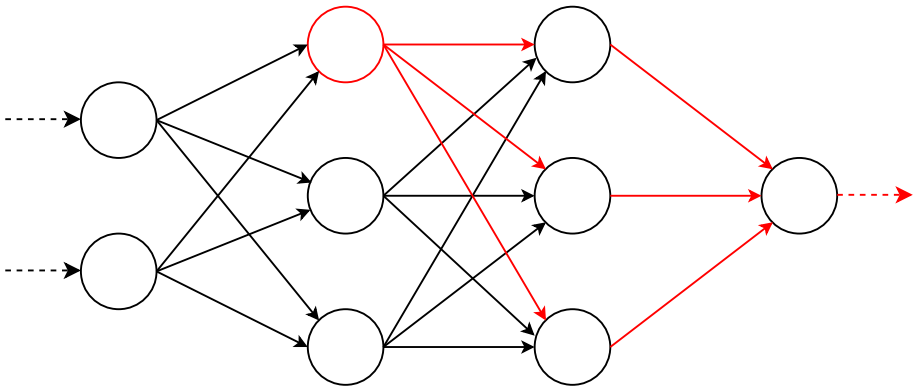


Figure 2.9: Simple fully connected neural network structure with two inputs and one output.

As seen, if a parameter value is updated in the node circled in red, that change propagates through the network and changes the output of every following node. As such, every parameter value in the network have to be updated simultaneously rather than independently in order to minimize the prediction error. To accomplish this, gradient based optimization techniques are used.

2.2.9.1 Gradient descent

Gradient descent is a class of optimization techniques that utilizes the derivative of the optimization loss with respect to the parameters that are being optimized. A step in the parameter values is then taken in the direction that minimizes the loss. In the context of neural networks, this means first propagating through the network to calculate a prediction, which in turn yields an error. Then, the derivative of the error with respect to weights in all nodes is calculated. An update is then performed on the values of the network parameters. There are several variations of the gradient descent technique that offer different characteristics and the problem at hand often dictates which technique is the best.

2.2.9.1.1 Batch gradient descent

When using batch gradient descent, the entire training set is used at the same time to calculate the parameter updates. More specifically, the error is computed using the prediction for every training sample. Then, the parameter update is calculating by minimizing the loss for all training samples at once, which in practise means once per epoch. This is very computationally efficient, but can be problematic for very large data sets, as all training samples need to be kept in memory. Another issue with this approach is that there is a risk that the optimization stops in a local optima. Model updates are also done very infrequently, which can lead to slow training speeds for very large data sets.

2.2.9.1.2 Stochastic gradient descent

Stochastic gradient descent is in some ways the complete opposite of batch gradient descent. Here, the parameter update is carried out for a single training example at a time. By doing this, the network parameters are updated very frequently, which can lead to faster convergence for some problems. Furthermore, calculating derivatives for a single training example at a time introduces noise into the gradients. As a result, stochastic gradient descent does not have as much problems with local optima as the batch gradient descent technique.

Dealing with one training sample at a time is very computationally inefficient, which means that the training process may be slower than for other gradient descent techniques. Moreover, noise in the gradients can lead to a large variance in the prediction error of the model for different training epochs.

2.2.9.1.3 Mini-batch gradient descent

Mini-batch gradient descent is a compromise between batch and stochastic gradient descent. The technique updates parameters using small batches of the training set at once. By choosing batch sizes that fits the hardware's memory architecture perfectly very high computational efficiency can be reached while maintaining frequent model updates. Mini-batch gradient descent offers the same benefits and drawbacks as stochastic and batch gradient descent to varying degree depending on the batch size chosen. As a result of this, it is the gradient descent technique most commonly used when training neural networks.

2.2.9.2 Backpropagation

One prerequisite to use gradient descent methods in optimization is that the loss function is differentiable with respect to the parameters. Backpropagation, which is short for the backward propagation of errors, is a method to calculate these derivates for neural networks. The algorithm starts in the output layer with the prediction error for a given sample. Then, the chain rule for multivariate functions is used to iteratively calculate the derivative of the error for one layer at a time. This is repeated until the input layer is reached and the gradients for all the networks parameters are calculated.

2.3 Image transformations

Transforming time series data to different types of images can be a way to gain insightful knowledge about the underlying process that the data was sampled from. It has also been shown useful for classification tasks in various domains [8, 9, 10, 35, 11]. One major drawback of image transformations is that the dimensionality of the input increases significantly, which may affect the computational complexity of the algorithm and convergence of the training procedure. Even so, image transformations can be used to extract discriminatory features that may improve classification performance. Two such transformations are Gramian angular fields (GAF) and recurrence plots (RP), which are both presented in the following sections.

2.3.1 Gramian angular fields

GAFs were proposed in [8] as a way to encode time series as images for use in a classification algorithm. To derive the Gramian matrices, the time series must first be scaled to range between 0 and 1 and then converted into polar coordinates. Given a time series x of N samples, the polar coordinates can be calculated according to

$$\begin{cases} \phi_i = arccos(x_i) \\ r_i = \frac{i}{N} \end{cases} \tag{2.15}$$

where ϕ and r is the angle and radius respectively. An example of this type of polar encoding can be seen below. An example time series signal is shown in figure 2.10. In figure 2.11 the polar encoding of said time series is depicted.

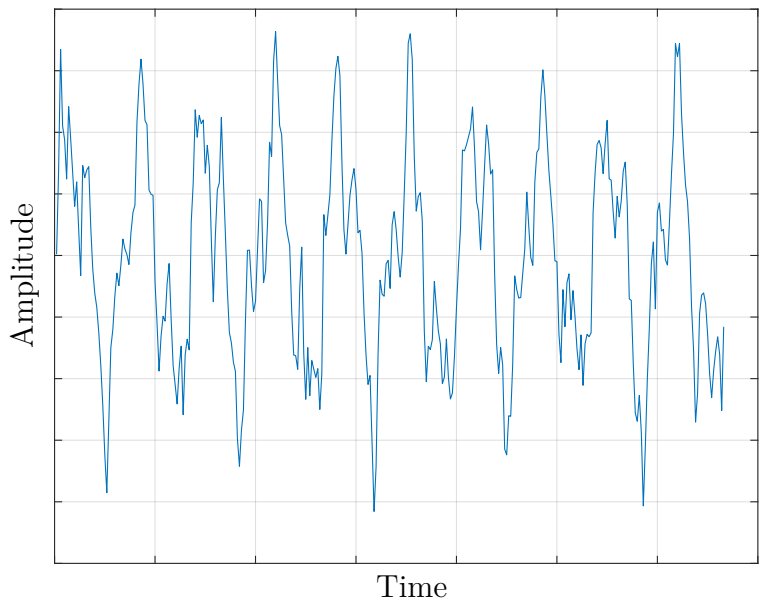


Figure 2.10: Example time series signal.

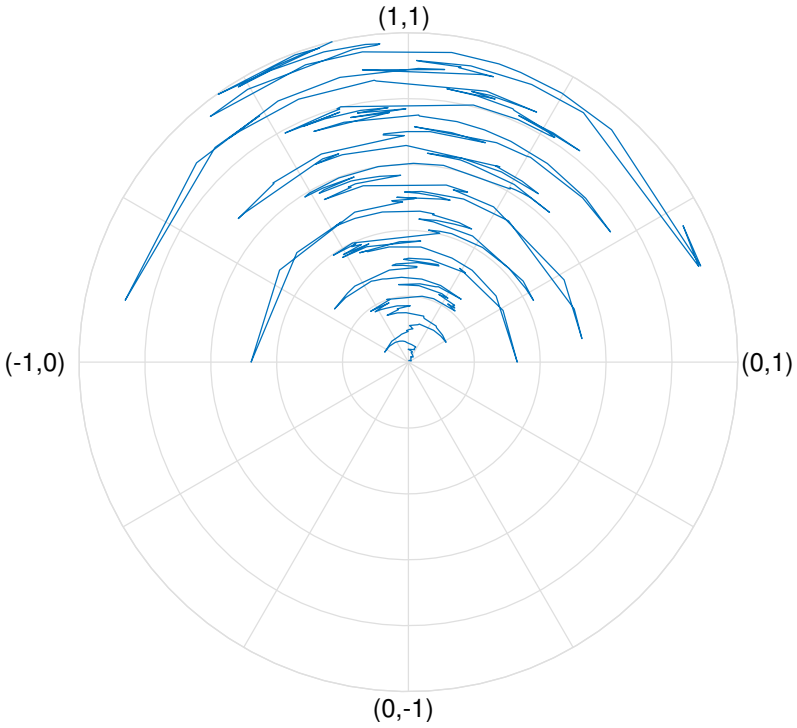


Figure 2.11: Unit circle with the polar encoding of an example time series signal.

With the polar encoding given, trigonometric sums and differences can be used to find dependencies between different time intervals in the signal. More specifically, the Gramian angular summation field (GASF) and Gramian angular difference field (GADF) are calculated according to

$$GASF = \begin{bmatrix} cos(\phi_1 + \phi_1) & \cdots & cos(\phi_1 + \phi_N) \\ cos(\phi_2 + \phi_1) & \cdots & cos(\phi_2 + \phi_N) \\ \vdots & \ddots & \vdots \\ cos(\phi_N + \phi_1) & \cdots & cos(\phi_N + \phi_N) \end{bmatrix} \tag{2.16}$$

$$GADF = \begin{bmatrix} sin(\phi_1 - \phi_1) & \cdots & sin(\phi_1 - \phi_N) \\ sin(\phi_2 - \phi_1) & \cdots & sin(\phi_2 - \phi_N) \\ \vdots & \ddots & \vdots \\ sin(\phi_N - \phi_1) & \cdots & sin(\phi_N - \phi_N) \end{bmatrix} \tag{2.17}$$

A visualization of GASF and GADF matrices calculated in this way can be seen in figures 2.12 and 2.13 respectively. These images are derived using the time series and polar encoding given in figures 2.10 and 2.11 above.

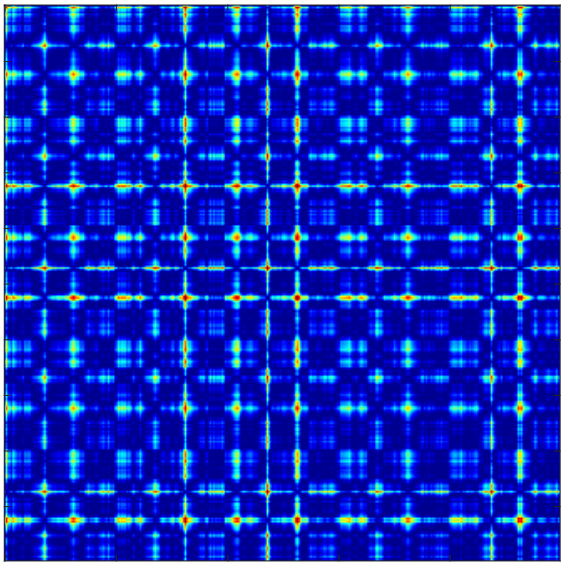


Figure 2.12: GASF for an example time series signal.

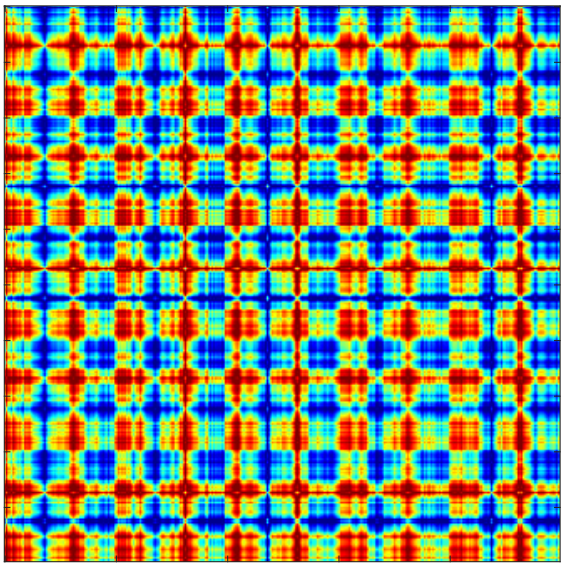


Figure 2.13: GADF for an example time series signal.

The transformations into GAFs are a way to preserve the temporal dependencies in the the time series data. Furthermore, the matrices yields the relative correlation by superposition and difference for varying time intervals. Lastly, the original time series can be recovered from the main diagonal of the GASF matrix, as it contains

the special case when the time interval is zero. Therefore, there is no information loss when going from a time series representation to the GAF representation.

2.3.2 Recurrence plots

RPs were originally proposed in [36] as a method to represent a phase space trajectory of arbitrary dimension as a two dimensional matrix. More specifically, a recurrence plot is a matrix that counts the number of recurrences of a dynamical system. A recurrence takes place whenever the phase space trajectory x gets within some distance to a point in the phase space that the trajectory crosses at some other time instance. Given a pair of time instances i and j, the corresponding element in a matrix of distances D can be calculated as

$$D(i,j) = \|x(i) - x(j)\| \tag{2.18}$$

Then, the recurrence matrix R can be acquired by filtering the distance matrix using a distance threshold ϵ, according to

$$R(i,j) = \begin{cases} 1, & D(i,j) < \epsilon \\ 0, & \text{otherwise} \end{cases} \tag{2.19}$$

These matrices can then easily be visualized. In figures 2.14 and 2.15 the distance and recurrence matrices for the example time series given earlier in figure 2.10 are shown.

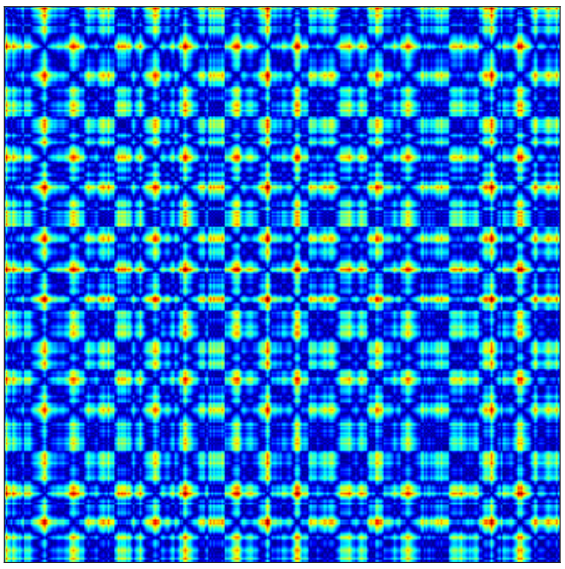


Figure 2.14: Distance matrix for an example time series signal.

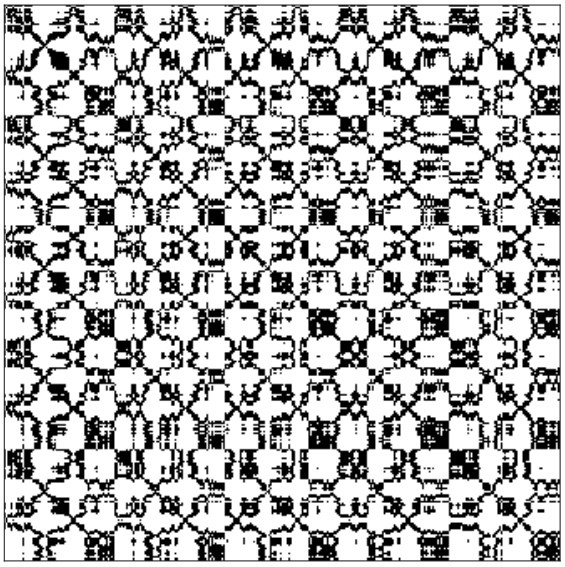


Figure 2.15: Recurrence matrix for an example time series signal.

This type of image transformations can be used to identify several aspects of the underlying process, such as periodicity and trends. As such, the recurrence transformation can be used to extract meaningful features from a time series as part of a classification or regression algorithm.

3
Method

3.1 Development setting

In order to efficiently develop and evaluate various neural network structures, various software tools are necessary. Python is the most commonly used programming language when it comes to data science and machine learning [37]. Several tools and libraries for developing machine learning algorithms exists, which can greatly accelerate the development process. In this work, the TensorFlow library for Python is used because of its extensive documentation and wide support for various neural network architectures [38]. Keras, which is a high level application programming interface for different machine learning libraries, will be used to interact with the TensorFlow backend [39]. The benefit of using Keras is that it unifies several machine learning libraries into a single syntax and way of working. This yields more interpretable code and allows for fast experimenting and prototyping of neural network models.

3.2 Data set

For the task of training models, a large data set of annotated samples is used. The input data is the sampled current consumption of the linear actuator that the roof deflector is mounted on. Each observation in the input data corresponds to a time series of current measurements taken while the linear actuator extracts from the minimum to maximum position. During every sweep, the trailer height and gap has been kept fixed and logged. The height and gap data has later been used to calculate the optimal actuator position, according to the table given by the truck models handbook. This set of optimal positions is what constitutes the output data to be used for training and evaluating models.

The data set has been collected using two separate truck models from different manufacturers. Furthermore, the mechanical configuration of the roof deflector is different for the two brands. The most major difference being that there is either one or two actuators moving the roof deflector, depending on the brand. In the dual case, there is more data available as the sampled current consumption is a series with two channels, one for each actuator. However, the aerodynamic load is now shared between the two actuators, which means the signal-to-noise ratio is significantly lower.

Due to the difference in mechanical configuration, there are large differences between the collected data for the two brands and training a single model that

performs well for both truck models is therefore highly unlikely. As such, it is desired to train separate models for the two models and only evaluate them on data from the same model.

This work primarily use data from the single actuator case. Both data sets are used during the development process in order to converge to a limited number of network architectures that work well for both truck brands. However, the final evaluation and comparison of networks will be done using the single actuator data for the sake of clarity. In the remainder of this report, all mentions of the data set is in reference to the single actuator data unless otherwise specified.

The available sweep data is collected while the truck is driving over a certain speed threshold. This is done to ensure that the small aerodynamic effects that the models aims to find are significant enough to be detected in the data. While above the threshold, there is no emphasis on keeping the vehicle speed fixed. This corresponds well to normal driving conditions, which is crucial for good generalization of the models trained. A histogram of the vehicle speed present in the data set used can be seen in figure 3.1 below.

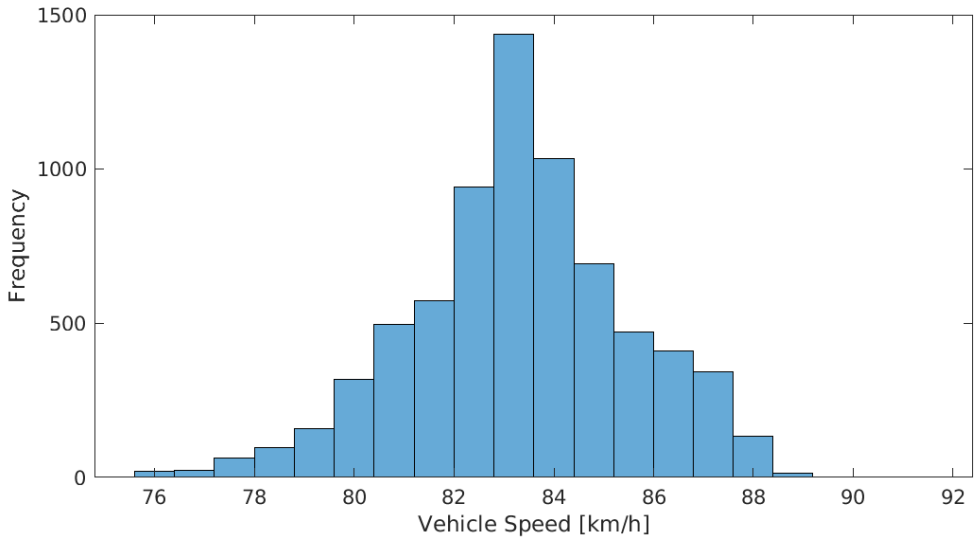


Figure 3.1: Histogram of the vehicle speed during the collected data set.

Furthermore, the height and gap configuration of the trailer is also varied throughout the data collection. This is done using an adjustable trailer that can be extended forwards and upwards while the truck is driving to simulate different trailer configurations. These height and gap configurations all correspond to an optimal roof deflector position. Histograms for the trailer height, trailer gap and optimal roof deflector position during data collection can be seen in figures 3.2, 3.3 and 3.4 respectively.

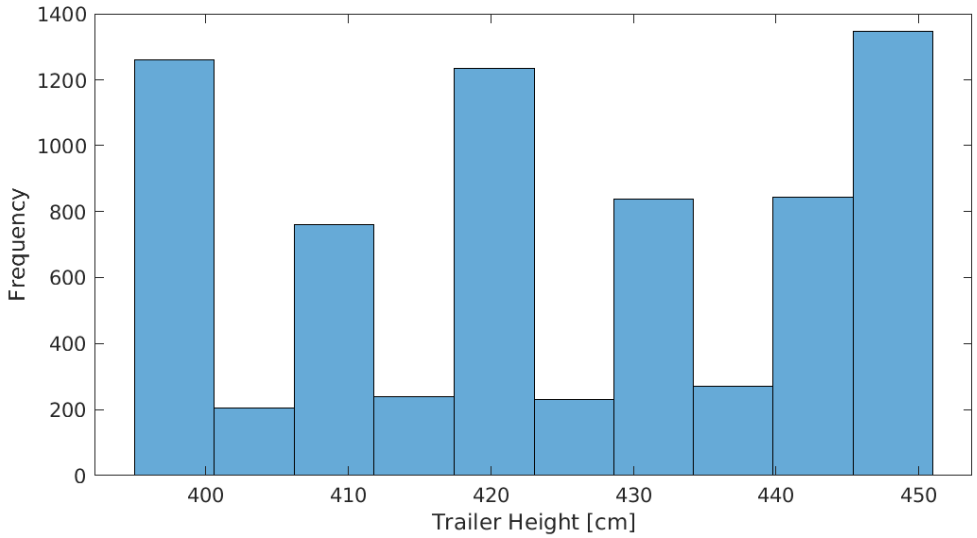


Figure 3.2: Histogram of the collected trailer heights in the data set.

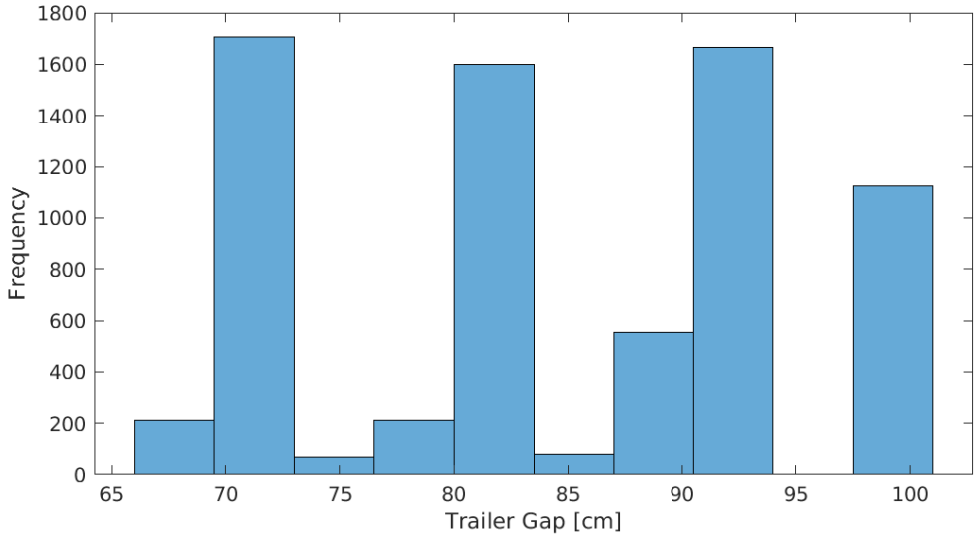


Figure 3.3: Histogram of the collected trailer gaps in the data set.

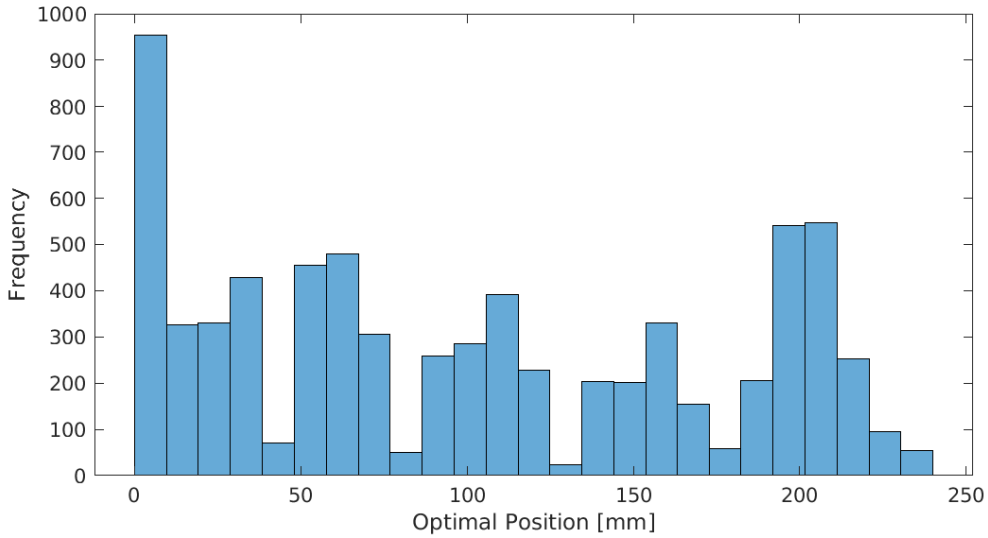


Figure 3.4: Histogram of the optimal roof deflection position in the data set.

It is important to note that a specific roof deflector position can be optimal for several different height and gap configurations. This means that the aerodynamic effects, and by extension the current consumption, can look significantly different even for the same optimal roof deflector position.

To summarize, the models will be trained and evaluated using series of current consumption data during sweeps carried out with a single linear actuator. Every time series have a length of 280 samples. Furthermore, each observation has a corresponding optimal position, which is a function of the trailer height and gap. In total, there are 7237 number of observations available for training and evaluation.

3.3 Network architectures

In order to evaluate the effectiveness of using neural networks for predicting the optimal roof deflector on trucks, several different architectures are implemented. One very important aspect of the classification problem at hand is the temporal characteristics of the input signals. It is therefore highly beneficial if the algorithms are able to model this kind of dependencies. Thus, convolutional neural networks are very suitable for this type of application, as they have been used in other domains to identify similar characteristics.

The focus of this section is to present the various neural network architectures that are implemented and evaluated. The complexity of each architecture can be tuned by increasing or decreasing the number of nodes in each layers. As there are infinitely many ways to choose the number of nodes, it is not feasible to evaluate every combination. Instead, for each type of architecture, a base complexity will be set. Then, variants of this network where a factor k has been used to scale the number of nodes in each layer will be evaluated. As an example, consider a base architecture that consists of 3 layers with 8, 16 and 8 nodes respectively. Using $k = 2$, a network with 16, 32 and 32 nodes in each layer can then be realized.

By scaling the networks in this way a comparison and evaluation of the networks complexity can be made while retaining the relative complexity between layers inside the networks.

3.3.1 Time series based networks

As the goal is to find an approach that offers complete end-to-end functionality, the natural first step is to investigate neural network approaches where feature extraction is learned automatically. As such, several convolutional neural networks that function as complete end-to-end algorithms are implemented and presented in this section. All the neural network architectures presented use raw current consumption sequences as input and any data processing steps are part of the learned algorithm.

3.3.1.1 FCNN

Fully convolutional neural networks (FCNN) were proposed in [5] as a baseline approach for any time series classification problem. The presented architecture consisted of three subsequent convolutional layers, followed by a global averaging pool layer. At the output of every convolutional layer, the signal is normalized using a batch normalization layer before being passed through a ReLU activation function. The convolution operation in each layer uses a stride of 1 and pads the input sequence in order to maintain the dimensions of the time series throughout the network. An illustration of the FCNN architecture can be seen in figure 3.5.

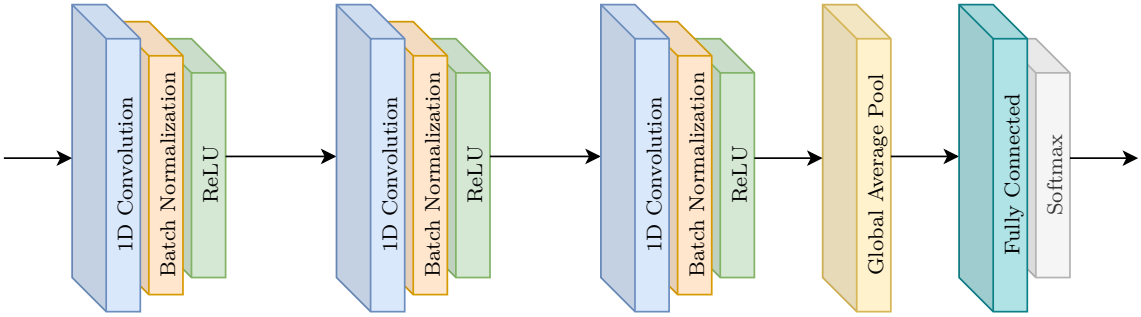


Figure 3.5: FCNN architecture.

The baseline for this architecture is to use 8, 16 and 8 nodes in the first, second and third convolutional layers respectively. These numbers are then adjusted using a scaling k to evaluate various complexities as discussed before.

3.3.1.2 LSTM-FCNN

In [6] it was shown that augmenting the FCNN architecture with an LSTM cell can be beneficial for performance. The cell processes a dimension shuffled version of the input in parallel to the FCNN structure. A dropout layer is added after the LSTM cell to avoid overfitting. The outputs from the two branches are then concatenated

before being passed to the output layer. The resulting architecture can be seen in figure 3.6 below.

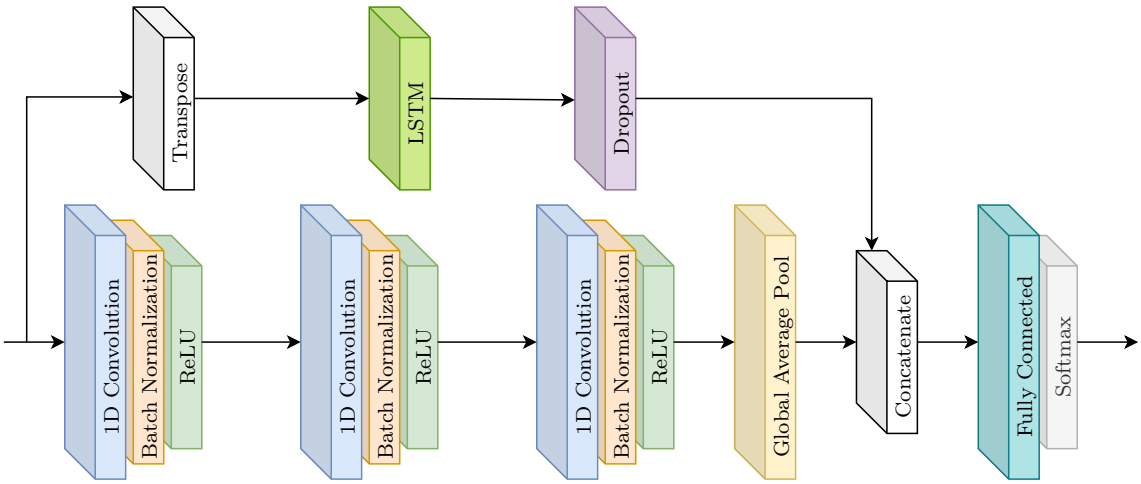


Figure 3.6: LSTM-FCNN architecture.

3.3.1.3 FCNN-SE

Another addition to the FCNN architecture was proposed in [40]. It introduces an SE block after the activation function of the two first convolutional layers. The SE block aims to capture channel-wise dependencies. Note that the input data in this application only has a single channel. However, the output of a convolutional layers consists of as many channels as there are neuron in the layer, which enables the application of an SE block. The resulting architecture when using the SE augmentation on the FCNN architecture can be seen below in figure 3.7.

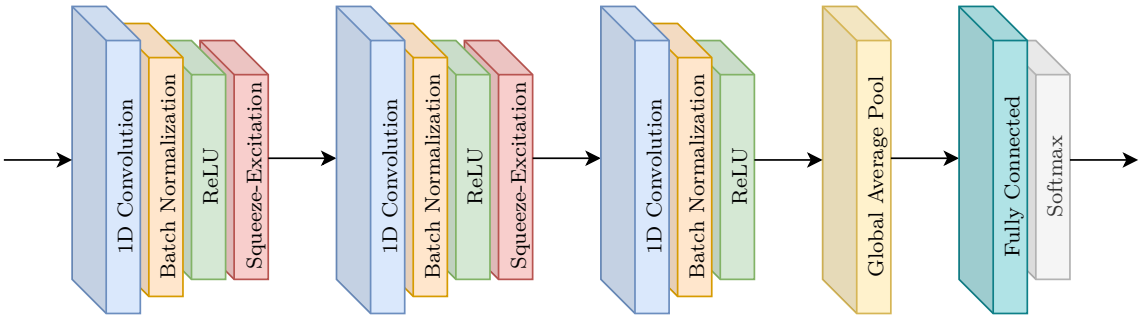


Figure 3.7: FCNN-SE architecture.

3.3.1.4 LSTM-FCNN-SE

The SE augmentation can also be applied on the LSTM-FCNN architecture. The resulting architecture when doing this can be seen in figure 3.8 below.

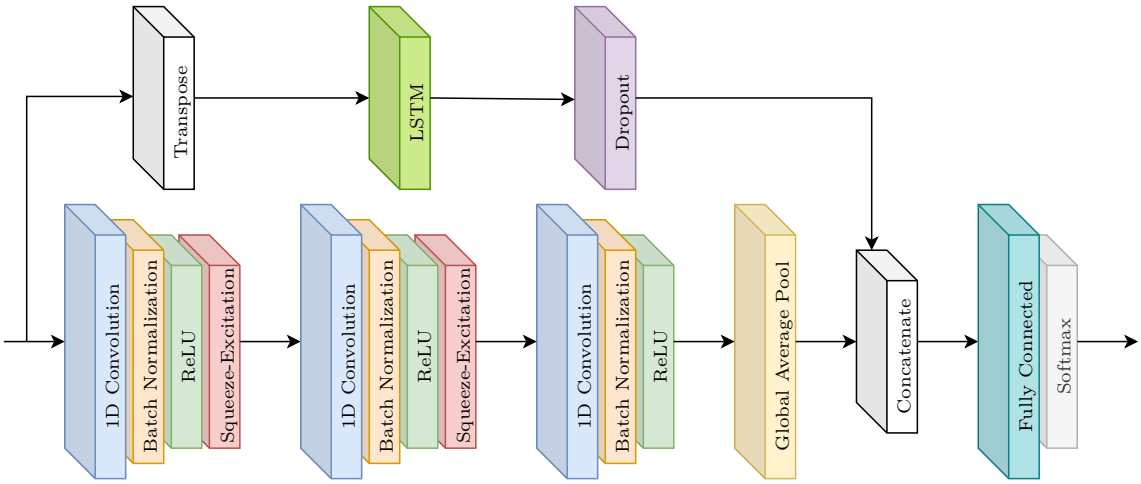


Figure 3.8: LSTM-FCNN-SE architecture.

3.3.2 Image based networks

Another approach to the time series classification problem is to use image transformation to extract useful features for distinguishing between classes. As such, a neural network architectures that utilize GAFs and RPs as input data have been implemented. The image transformations are not a part of the trainable network, but they are an essential component for the end-to-end functionality of the proposed prediction model. The implemented image transformation procedure is depicted in figure 3.9 below.

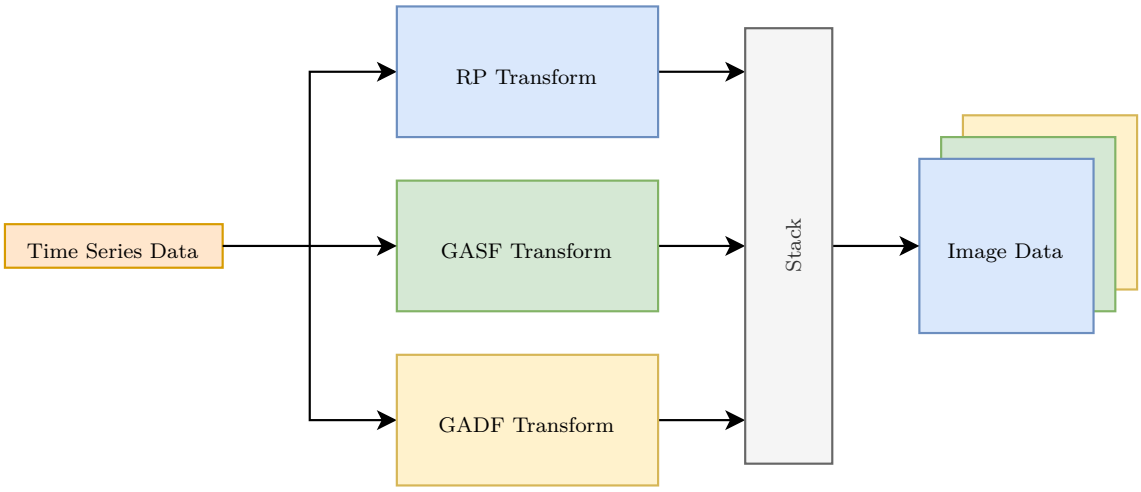


Figure 3.9: Time series to image transformation procedure.

As can be seen, a time series in the input time series generates 3 channels of image data. The GASF, GADF and RP images are then stacked in the third dimension before being sent as input to an image based neural network with both convolutional and fully connected layers (IM-CNN). The size of the images can be scaled freely depending on the desired level of detail in the images. As such, networks trained

on images with sizes of 64 and 128 are evaluated to see how much resolution affects the performance and computational complexity.

3.3.2.1 IM-CNN

The IM-CNN architecture uses a combination of convolutional and fully connected layers. The image inputs are first processed by the convolutional layers, and then extracted features are then handled by fully connected layers in order to produce a prediction. Similar to previously presented architectures, batch normalization and the ReLU activation function is used after each convolution. Furthermore, a max pooling operation is inserted after each convolutional layer with a pooling size of 2. This reduces the size of the feature images to half after each convolutional layer, which in turn reduces the number of parameters in subsequent layers. Similar to the convolutional layers, the fully connected layers also utilize batch normalization and ReLU activation functions. The complete IM-CNN architecture can be seen in figure 3.10 below. Note that the architecture is the same for the network based on images of size 64 (IM64-CNN) and the network based on images of size 128 (IM128-CNN).

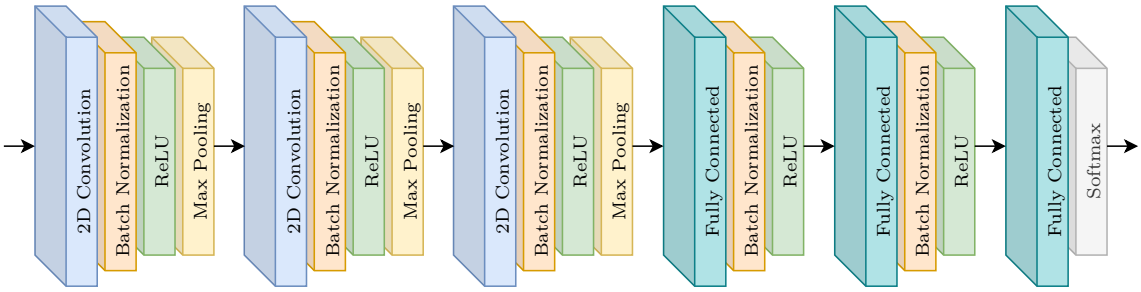


Figure 3.10: IM-CNN architecture.

The baseline complexity for the IM-CNN architecture is to use 8, 16 and 8 nodes in the convolutional layers. Furthermore, the fully connected layers both have 8 nodes each.

3.4 Evaluation metrics

Several evaluation metrics are necessary in order to objectively compare the derived neural network in various aspects. The predictive performance is evaluated using a function that relates the error in estimated optimum position to a relative increase in fuel consumption. Number of parameters in each model will be used as the metric for comparing architectural complexity. In the following sections, the mentioned evaluation metrics are broken down and outlined in more detail.

3.4.1 Fuel consumption penalty

Fuel consumption penalty (FCP) is a way to transform the prediction errors of an algorithm into something more tangible that relates to the fuel consumption of the truck. It makes requirements set on the end product directly related to the

algorithm which help the development and evaluation process. It also summarizes the predictive performance on a whole data set into a single value.

The ground truth for prediction is derived using the truck manufacturers handbook for setting the roof deflector. This typically consists of taking some measurements of the cab and trailer and then looking up a value in a table in the handbook. The optimal position is not a single number, but rather a small range, bounded by $\pm\epsilon$. Inside this range, the position is considered optimal and thus yields no penalty to the fuel consumption. Outside the optimal position range, the increase in fuel consumption can approximated by straight lines. For roof deflector positions higher than the optima the slope is generally slightly steeper than for positions lower than the optima. However, for practical purposes these slopes can be averaged to form the slope coefficient K. This K constitutes the percentual increase in fuel consumption per millimeter in error. The exact values of the limit ϵ and the slope K depend on the truck and roof deflector used.

To produce a single value for an entire set of test examples, the FCP is calculated as the mean of the fuel consumption increase over all observations. For clarity, the calculation of FCP can be summarized into a mathematical expression. Given the vector of N prediction errors $e = |y - \hat{y}|$, the FCP value for the set can be calculated as

$$FCP(e) = \frac{1}{N} \sum_{i=1}^{N} \left[\begin{cases} 0, & e_i < \epsilon \\ K(e_i - \epsilon), & \text{otherwise} \end{cases} \right] \tag{3.1}$$

In the evaluation, the FCP values are normalized, FCP_{norm}, against a baseline algorithm trained on the same data set according to

$$FCP_{norm} = \frac{FCP}{FCP_{baseline}} \tag{3.2}$$

where $FCP_{baseline}$ is the baseline algorithms FCP on test data. The baseline algorithm is not based on deep neural networks and instead uses classical machine learning. As a result of the normalization, the baseline FCP after normalization is 1. The normalized FCP can be used to easily determine if the neural networks perform better or worse than the baseline and by how much. E.g. a normalized FCP of 0.8 corresponds to a 20% decrease in FCP from the baseline and a normalized FCP of 1.1 corresponds to an increase in FCP by 10%.

3.4.2 Number of parameters

Counting the number of parameters in a neural network gives an estimate of how much space is required to store the trained model. Furthermore, it can be a measure of complexity in the case of similar network architectures. In this count, both trainable and fixed parameters of the neurons in the network is included. Even so, this number can not accurately be transformed into a measure of storage space in bytes, as this is largely dependent on various implementation aspects. However, the actual parameters of a network is by far the biggest contributor to storage requirements when it come to deployment. Any overhead used to describe the architecture, operations or connections is insignificant in comparison. Performing a full analysis of every neural network is a very time consuming process and does

not yield much useful information in a direct comparison. For practical purposes, the number of parameters is therefore a good enough metric to use in evaluation of different neural networks. The number itself is often largely decided by the number of fully connected or convolutional layers and their depth, as other types of layers contain very few parameters in comparison.

The number of parameters NOP_{FC} in a fully connected layers can be calculated using the input size n_{in} and the width of the layer w, according to

$$NOP_{FC} = w(n_{in} + 1) \tag{3.3}$$

In the case of multidimensional inputs, n_{in} denotes the total number of elements in the matrix across all dimensions. This is because all inputs are vectorized before being passed to a fully connected layer.

For a convolutional layer, calculating the number of parameters is slightly different. It also differs for time-series and image input data, due to varying number of dimensions in the convolution. Using the same notation as before, with the input size n_{in} and width of the layer w, the number of parameters for a convolutional layer NOP_{CNN} can be calculated as

$$NOP_{CNN} = \begin{cases} w(k_w c + 1), & \text{for time series inputs} \\ w(k_w k_h c + 1), & \text{for image inputs} \end{cases} \tag{3.4}$$

where k_w is the filter kernel width, k_h the kernel height and c is the number of channels in the input.

In LSTM layers, the number of parameters NOP_{LSTM} can be calculated using input dimension n_{in} and the number of hidden units w in the layer according to

$$NOP_{LSTM} = 4(w^2 + w(n_{in} + 1)) \tag{3.5}$$

for both time series and image input data.

3.5 Evaluation strategy

To evaluate the neural network based approach, a large number of neural networks are first trained using varying values of complexity k and dropout rate d. Then, a small selection of the most promising networks are chosen for further analysis. The predictive performance of all neural networks are measured as the percentual increase or decrease in FCP compared to the baseline algorithm. The methodologies of training, comparing and analyzing the networks are presented in the following sections.

3.5.1 Training

All evaluated neural networks are trained using the same training options and data set. The available data will be split in half in order to form a training set and a test set. The training is based on the Adam optimization algorithm proposed in [41] together with mini-batch gradient descent. FCP is used as the loss function

that optimizer attempts to minimize by iteratively adjusting the parameters of the networks. Because the Adam optimizer employs an adaptive learning rate scheme, it is used with its default initial learning rate setting of $\alpha = 10^{-3}$. A batch size of 256 is used, as it strikes a good balance between training speed and convergence for the optimization. The actual learning procedure is run for a total of 1000 epochs, i.e. passes over the training set. After each epoch, the training set is shuffled randomly as this has been shown to improve convergence of optimization based on mini-batch gradient descent based [42]. Note that the training and test splits are preserved and the training set is only shuffled internally. Moreover, the loss on the test data is calculated at the end of each epoch and if it is lower than for previous epochs, the current parameter values are saved externally. When the training procedure is complete, the parameter values that yielded the test loss are loaded from the external file and used for evaluation. By doing this, an early stopping criteria is not necessary and any problems of overfitting due to prolonged training are avoided.

Furthermore, dropout layers will be used in the networks so that the more complex architectures are able to reach acceptable performance and avoid overfitting. The dropout layers are inserted after the activation function of each convolutional layer in the time series based networks, and after the fully connected layers in the image based networks. The already existing dropout layer in the LSTM networks, seen in figures 3.6 and 3.8, is kept at a constant rate of 0.8 as suggested by the authors in [6]. Note that the addition of dropout only impact the training procedure and most importantly has no affect on the complexity or number of parameters in the network.

Both the complexity scaling k and dropout rate d are varied in order to find the best performing network for each architecture using a grid-search approach. The values of k and d used for this are given below.

$$k = [1,\ 2,\ 4]$$

$$d = [0,\ 0.2,\ 0.4,\ 0.6,\ 0.8]$$

A unique network is trained and evaluated for each configuration of network architecture, k and d. As a result, a total of 90 trained networks with six different architectures are subsequently produced.

3.5.2 Comparison

From the complete pool of 90 networks, the best performing network of each complexity and architecture is then selected for comparison. The networks are compared against each other in terms of predictive performance and complexity. In other words, both normalized FCP and NOP are used. By doing this comparison, the number of networks can be reduced into something manageable for further analysis. This smaller subset of networks will contain the best performing networks for varying levels of complexity.

3.5.3 Analysis

The purpose of reducing the number of networks is to enable the possibility of retraining the networks to evaluate the effect of different data set characteristics.

Because of the large amount of time it takes to train multiple networks, it is not feasible to conduct this type of analysis on every network.

First, it will be analyzed if aggregating data into batches during testing has an affect on performance. Due to the nature of the application, it is possible to collect multiple sweep signals in a row while knowing that all of them belong to the same optimal position. The batch can then be processed by an algorithm to form a prediction based on information from all of the individual sweep. This approach has proven to be successful in improving the performance of traditional machine learning techniques in this application. As such, it will be investigated if this is the case for the neural networks as well. To leverage the information of all sweeps in a batch, the networks will predict the optimal position for each individual sweep. Then, the individual sweep predictions are averaged to form a final batch prediction. When using the data set in this type of batches, it is important to keep track of the ordering of the sweeps. To mimic the behaviour in the real application, the individual sweeps in a batch should be sequential. This affects the data splitting procedure substantially, as sweeps have to be aggregated into batches before being split into training and test sets. Furthermore, there will be some loss of data when building the batches, as there exist observations that do not have neighboring sweeps to form a batch. As such, to evaluate the effects of using batches during test, the networks have to be retrained with a data set that was generated with a specific batch size in mind. The batch sizes to be evaluated are three and five, which generates data sets with 6033 and 5155 observations respectively. For reference, the original data set contains 7237 observations, which means the set has shrunk significantly. These sets are split equally into training and test data as before.

Another aspect to be investigated is if the size and split of the data set affects the predictive performance notably. This is accomplished by retraining the networks on new splits of the original data set. Varying the size of the training data and plotting the predictive performance as a function of training set size, can indicate if gathering more data is a way to improve performance. This type of plot is called learning curve. As this requires retraining the networks multiple times, this analysis will use the small selection of promising networks as previously defined. For producing the learning curves, 80% of the original data set is used for training and 20% is used for testing. This yields less test observations, which may negatively affects the statistical significance of the test results. However, the size of the data set used in this project is sufficiently large to still yield acceptable results in this regard. The first point in the learning curves will use 10% of the training set observations. Then, increments of 10% are used until the network has been trained on the full training set. This results in a total of ten steps in the learning curve. Note that after each step, the network is used on the full test set to compute the corresponding FCP. Furthermore, after each step in the learning curve, the networks are reset to their untrained state. Because of the many iterations necessary to produce learning curves, the number of training epochs for each step is reduced to 500, instead of the original 1000. In other words, to produce one learning curve a network is trained for 500 epochs for a total of ten times using varying fractions of the available training data.

Finally, it will be evaluated if the stochasticity of the data splitting procedure

has had a large influence on the performance numbers achieved. The same small selection of network as before will be retrained an additional three times, but with new random splits of the data set. Doing three new permutations is not enough for any statistical evidence of the performance. However, it can be used to determine if the original data split was particularly biased by analyzing the spread in FCP values for different data splits.

4
Results

4.1 Training history

In this section, the training history of each network is presented. Figures 4.1 through 4.6 show the evolution of the loss function after normalization as the number of training epochs increases. Each plot shows the normalized FCP of a single network on both training and test data. The plots are grouped according to the network architecture used. Each column represents a given network complexity, indicated by k. Furthermore, each row represents a given dropout rate d.

FCNN

Training histories for the networks based on the FCNN architecture can be seen in figure 4.1 below.

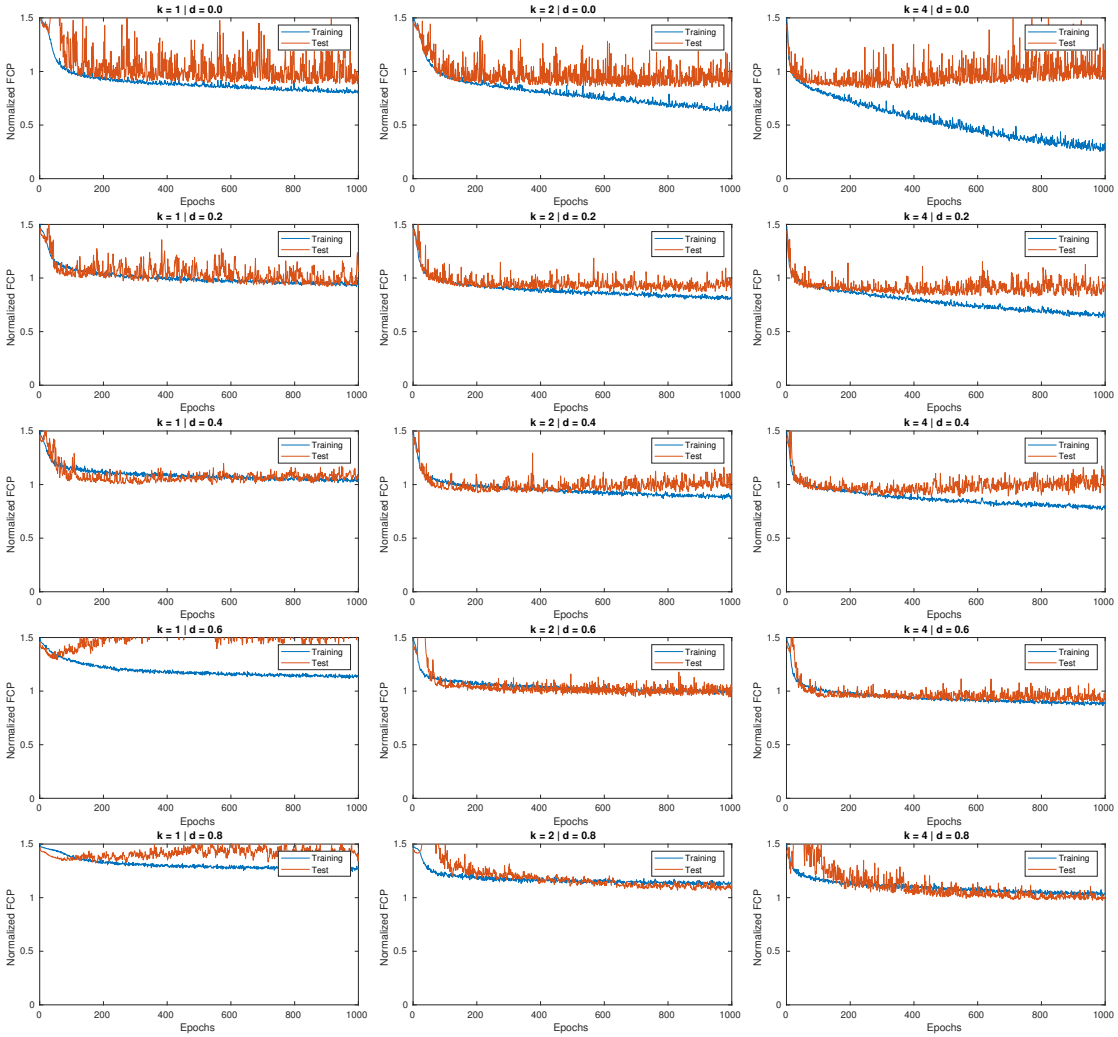


Figure 4.1: Training history for networks based on the FCNN architecture.

It can be seen that very few of the networks with the FCNN architecture manages to perform better than the baseline. In other words, the normalized FCP is lower lower than 1. With increasing k, the performance increases minimally. However, substanstial overfitting occurs in the networks with a high complexity k and low dropout rate d. The tendencies of overfitting are reduced with increasing d, but the performance does not become better.

LSTM-FCNN

Training histories for the networks based on the FCNN architecture can be seen in figure 4.2 below.

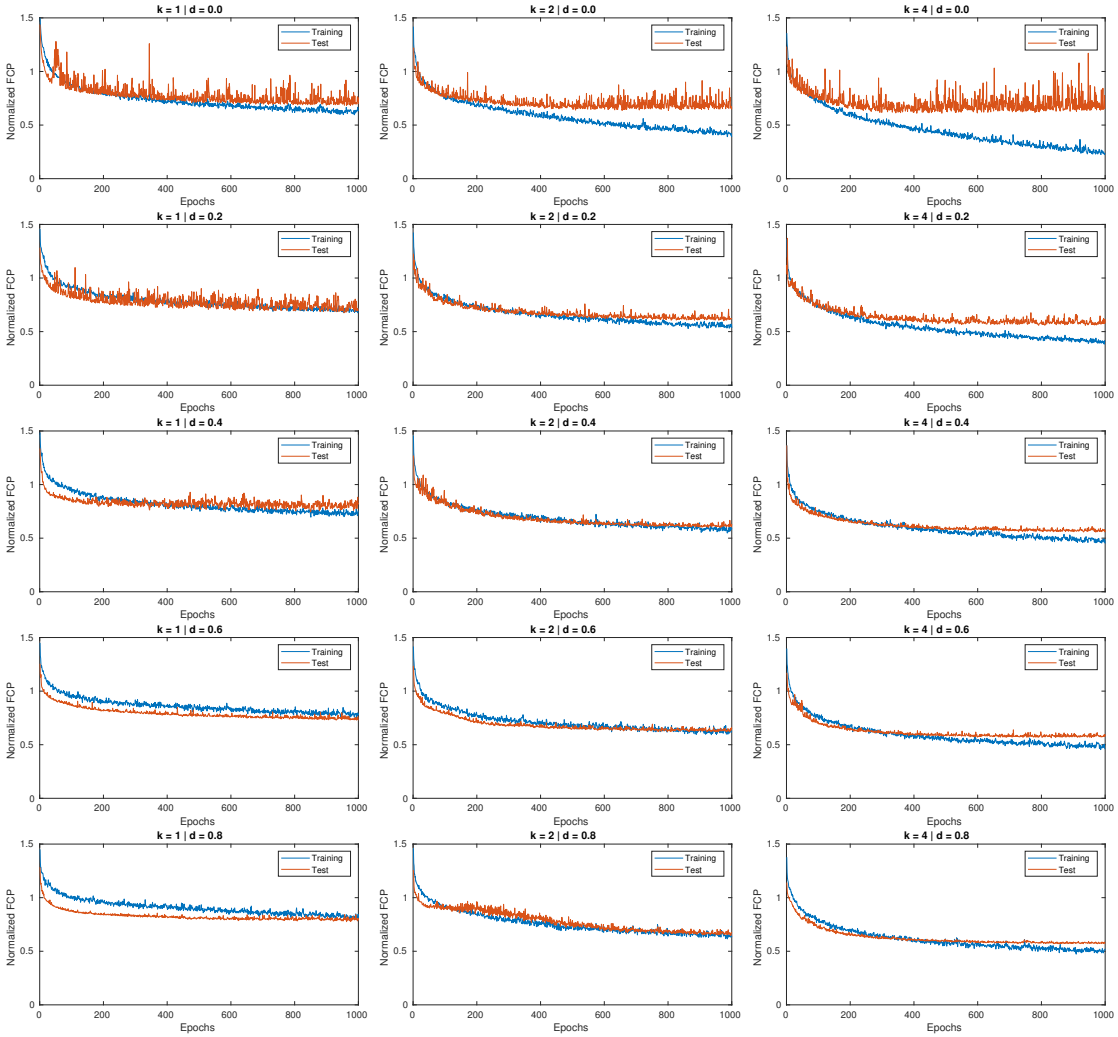


Figure 4.2: Training history for networks based on the LSTM-FCNN architecture.

The LSTM-FCNN architecture yields more promising results than the FCNN networks. All of the training histories reach lower FCP values than networks seen previously. Performance is generally better for higher complexity k. This is particularly clear when the dropout rate d also is high, as the networks overfit a lot less.

FCNN-SE

Training histories for the networks based on the FCNN architecture can be seen in figure 4.3 below.

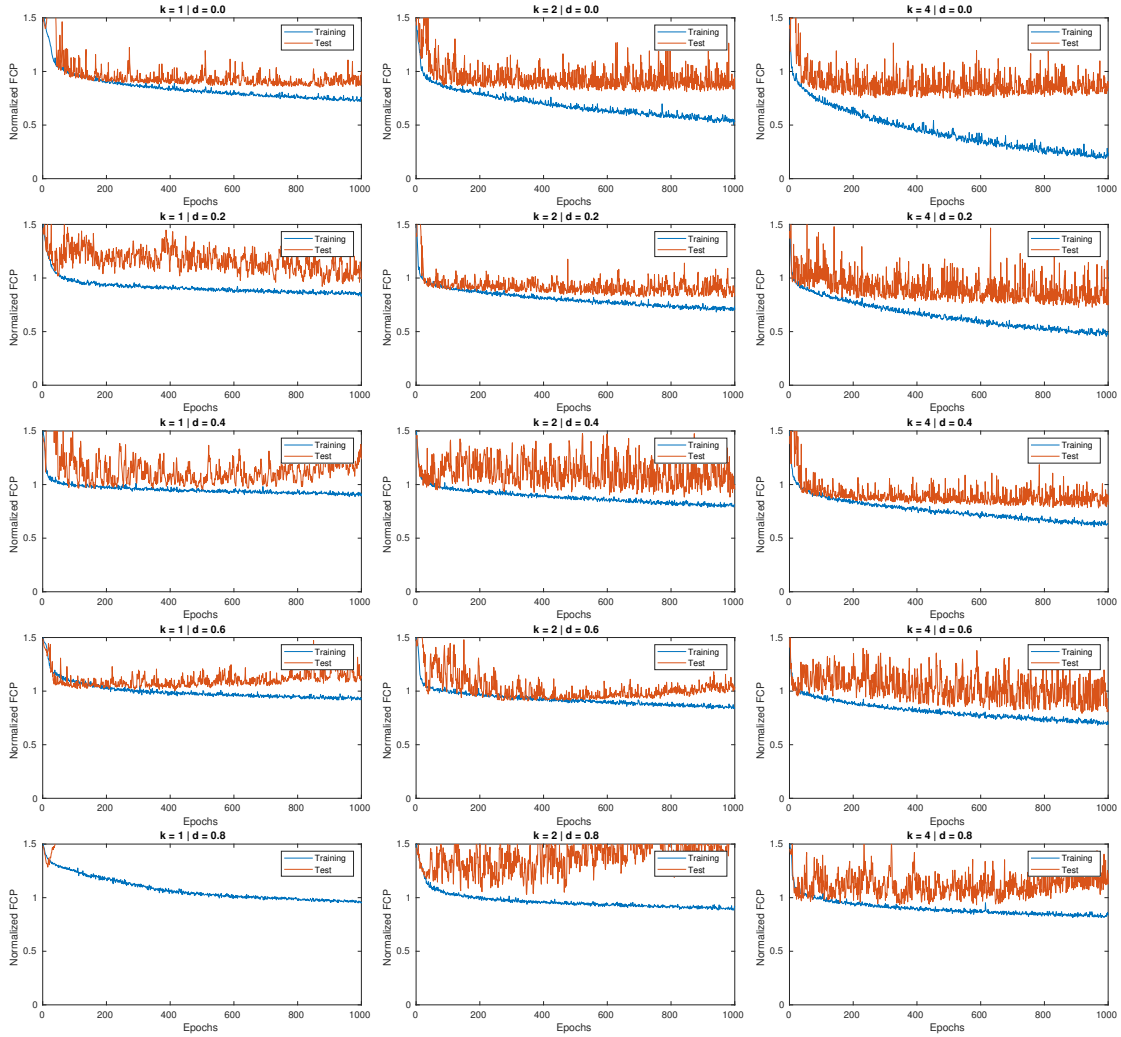


Figure 4.3: Training history for networks based on the FCNN-SE architecture.

The FCNN-SE architecture exhibits similar problems to what was seen for the FCNN networks. The FCP reaches slightly lower values than the FCNN case, but it is generally higher than the LSTM-FCNN results for most networks. Overfitting is reduced with increasing dropout rate d but does not give any improvements to the performance.

LSTM-FCNN-SE

Training histories for the networks based on the FCNN architecture can be seen in figure 4.4 below.

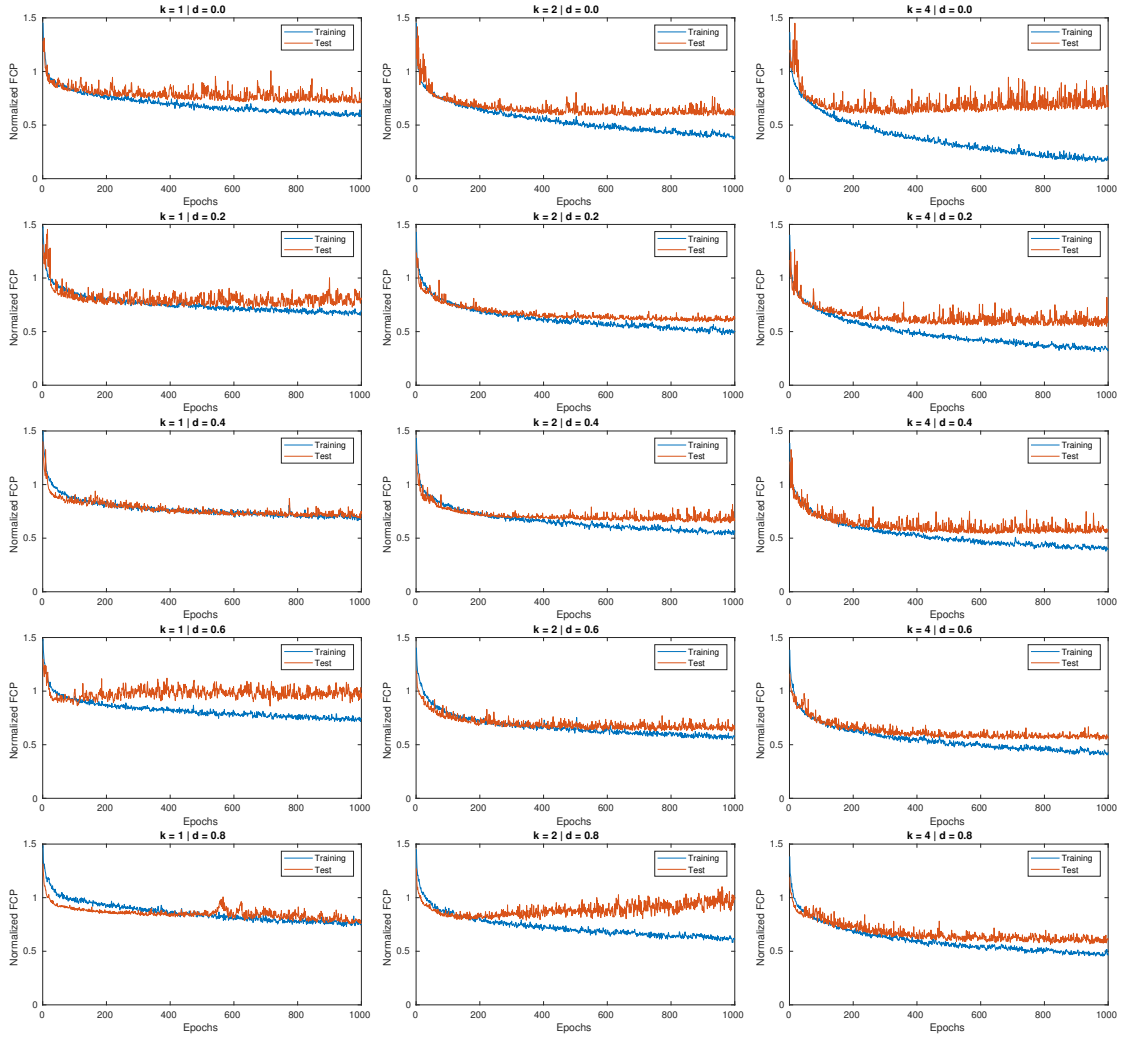


Figure 4.4: Training history for networks based on the LSTM-FCNN-SE architecture.

Networks with the LSTM-FCNN-SE architecture perform well compared to the previously seen network types. The FCP generally decreases with increasing complexity k. The increasing dropout rate d manages to reduce the overfitting and yields a clear improvement in the performance for the high complexity networks.

IM64-CNN

Training histories for the networks based on the IM-CNN architecture with input images of size 64 can be seen in figure 4.5 below.

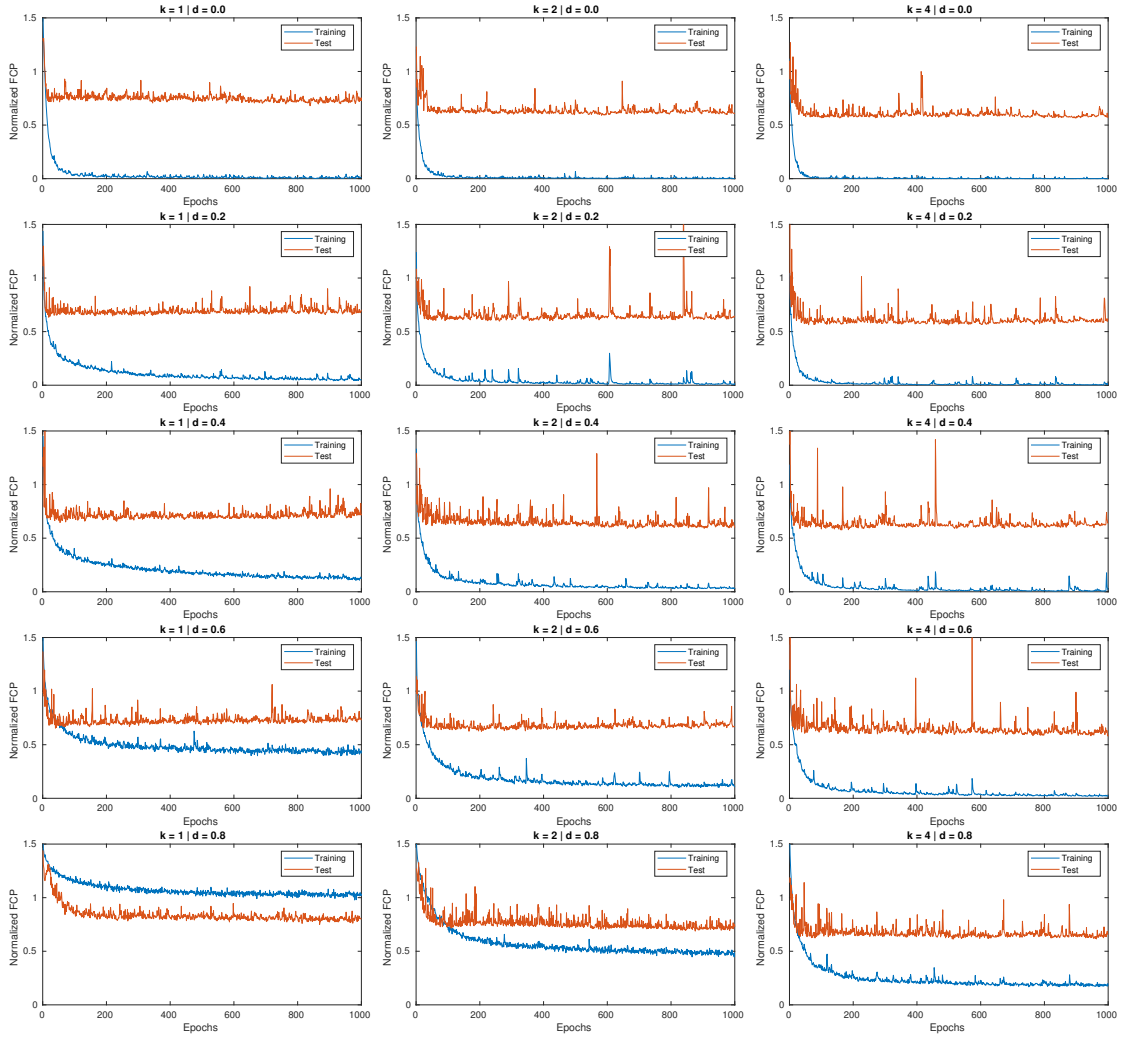


Figure 4.5: Training history for networks based on the IM64-CNN architecture.

It is clear that most networks with the IM64-CNN architecture have severe issues with overfitting. However, validation performance is quite good and on par with the LSTM-FCNN-SE networks even if the training loss is minimized very quickly. Increasing the dropout rate has a clear impact on the training loss trajectory, but unlike the other architecture it actually increases the overall validation loss.

IM128-CNN

Training histories for the networks based on the IM-CNN architecture with input images of size 128 can be seen in figure 4.6 below.

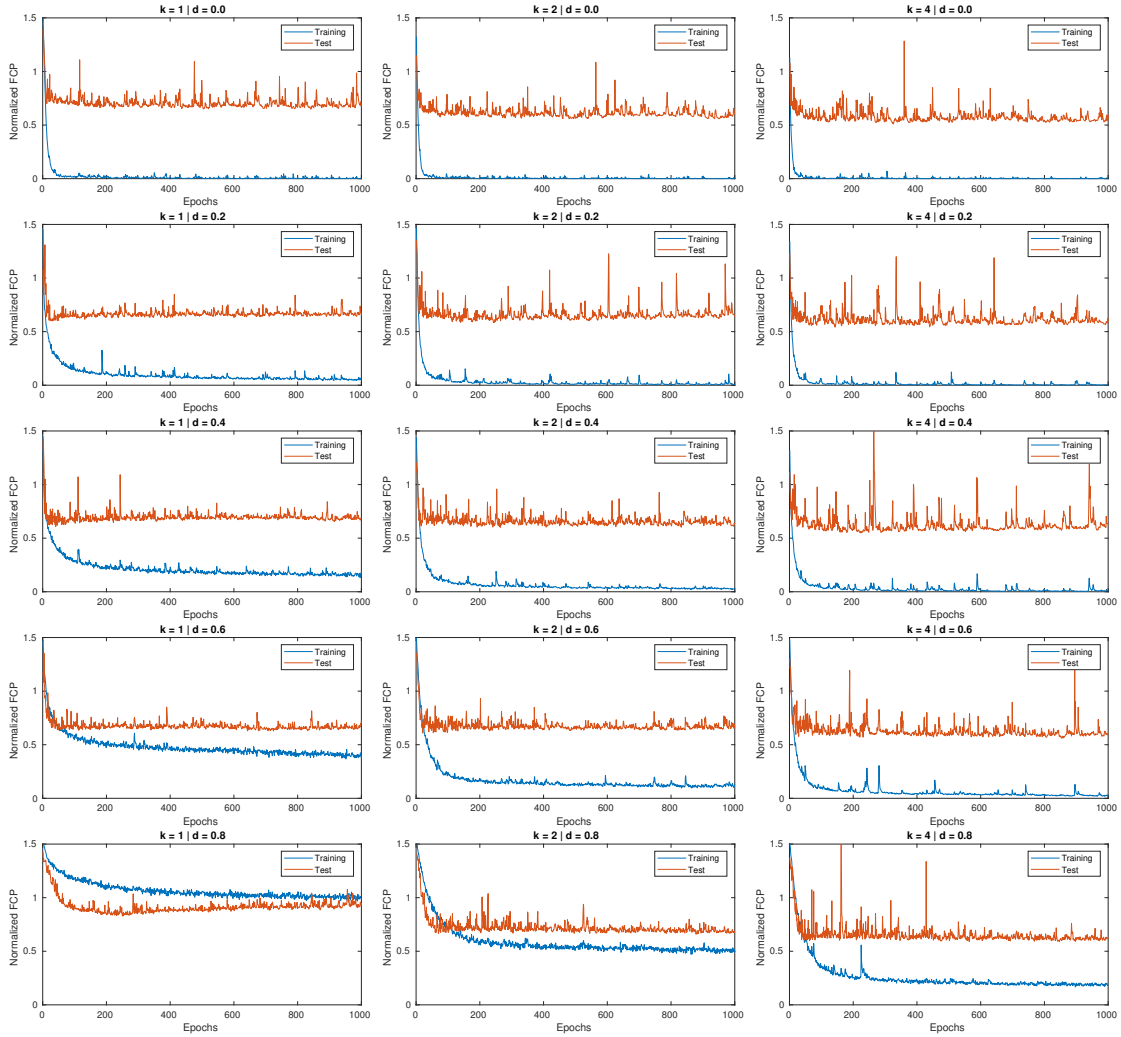


Figure 4.6: Training history for networks based on the IM128-CNN architecture.

The networks trained with images of size 128 have very similar tendencies as the ones trained on images of size 64. Overfitting occurs for networks of all complexities and is reduced significantly with increasing dropout rate. However, the reduction of overfitting does not yield any gain in predictive performance.

4.2 Predictive performance

The final performance, in terms of normalized FCP, for each of the trained networks can be seen in table 4.1 below. The final FCP value shown here is given by the lowest point in each of the validation loss curves seen in figures 4.1 through 4.6 previously.

Table 4.1: Final normalized FCP for all trained neural networks. The best FCP score for each architecture and complexity is shown in bold.

	$d = 0.0$	$d = 0.2$	$d = 0.4$	$d = 0.6$	$d = 0.8$
FCNN					
$k = 1$	**1.0051**	1.0474	1.1386	1.4747	1.4981
$k = 2$	**0.9693**	0.9883	1.0477	1.0736	1.2177
$k = 4$	0.9612	**0.9426**	1.0035	1.0202	1.1109
LSTM-FCNN					
$k = 1$	0.7771	**0.7705**	0.8479	0.8284	0.9941
$k = 2$	0.7254	0.6856	**0.6814**	0.7073	0.7363
$k = 4$	0.6953	0.6364	**0.6346**	0.6426	0.6487
FCNN-SE					
$k = 1$	**0.9706**	1.0506	1.0852	1.1445	1.4639
$k = 2$	**0.9328**	0.9301	1.0064	1.0340	1.1746
$k = 4$	0.8556	**0.8283**	0.8902	0.9033	1.0629
LSTM-FCNN-SE					
$k = 1$	0.8022	0.8097	**0.7769**	0.9800	0.8640
$k = 2$	**0.6608**	0.6723	0.7300	0.7124	0.8996
$k = 4$	0.6771	0.6328	**0.6141**	0.6181	0.6428
IM64-CNN					
$k = 1$	0.7746	**0.7223**	0.7358	0.7458	0.8543
$k = 2$	0.6731	0.6810	**0.6711**	0.7121	0.7834
$k = 4$	0.6436	**0.6404**	0.6564	0.6622	0.6994
IM128-CNN					
$k = 1$	0.7398	**0.6833**	0.6937	0.7150	0.9440
$k = 2$	**0.6364**	0.6660	0.6847	0.6942	0.7518
$k = 4$	**0.5812**	0.6168	0.6249	0.6397	0.6701

For all the trained network architectures, the performance increases with the complexity factor k. Whether the dropout rate d should be set high or low to maximize performance depends entirely on the networks architecture. When taking all the networks into account there are no clear trends that indicate that dropout has to be used to avoid overfitting for good performance with high complexity networks. On the contrary, the best performing network in this experiment is the IM128-CNN architechture with complexity factor $k = 4$ and dropout rate $d = 0$, which clearly overfit the training data as seen in figure 4.6. However, for time based networks with the LSTM augmentation and high complexity, a higher dropout rate have a clear positive impact on the performance.

Looking at the final FCP value for all time series based networks it is clear that the LSTM augmentation performs very well. Both LSTM-FCNN and LSTM-

FCNN-SE perform significantly better overall compared to the FCNN and FCNN-SE counterparts. The SE augmention also reduces the final FCP values, especially when not combined with the LSTM augmentation. While the FCNN-SE networks have clearly better performance than the FCNN networks, the SE augmentation on its own is not enough to reach the performance standards set by the LSTM-FCNN and LSTM-FCNN-SE networks.

When it comes to the image based networks, using a higher resolution image yields a notable performance increase. Overall, both IM64-CNN and IM128-CNN are on par with the best performing time series networks, LSTM-FCNN and LSTM-FCNN-SE, in terms of final FCP values.

4.2.1 Cumulative distribution function

Another way to examine the predictive performance is to study the cumulative distribution function (CDF) of the prediction errors. Rather than a single number as the metric of performance, the CDF plots indicates what types of errors the model makes. More specifically, the CDF plot shows the percentage of prediction errors that are within certain ranges. A CDF plot for the networks trained in this work can be seen in figures 4.7 through 4.12 below. Note here that the error axes are all normalized against an arbitrary maximum value. However, all figures are normalized against the same value to allow for comparison of maximum error between figures. For clarity, only the best performing network of each complexity, i.e. the networks marked with bold in table 4.1 above are shown in the CDF plots below.

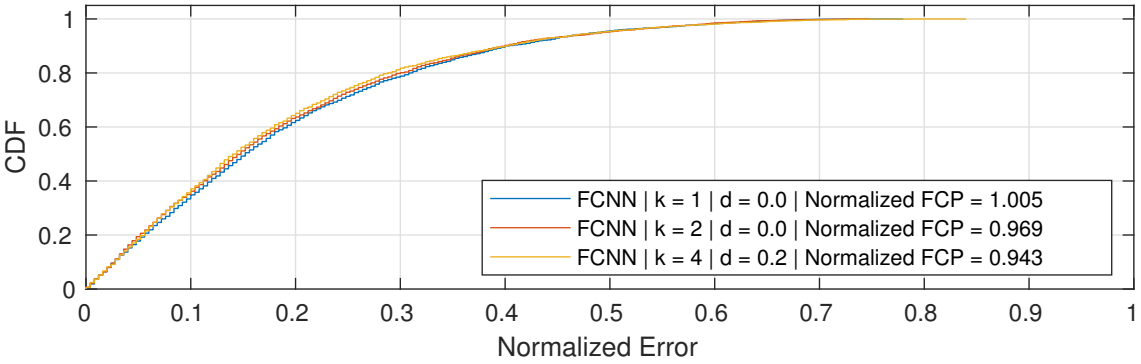


Figure 4.7: CDF curves for the best performing FCNN network for each level of complexity.

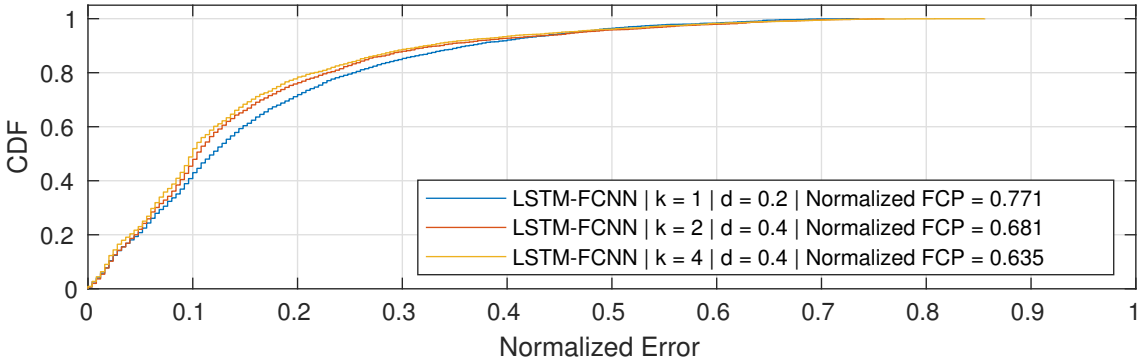


Figure 4.8: CDF curves for the best performing LSTM-FCNN network for each level of complexity.

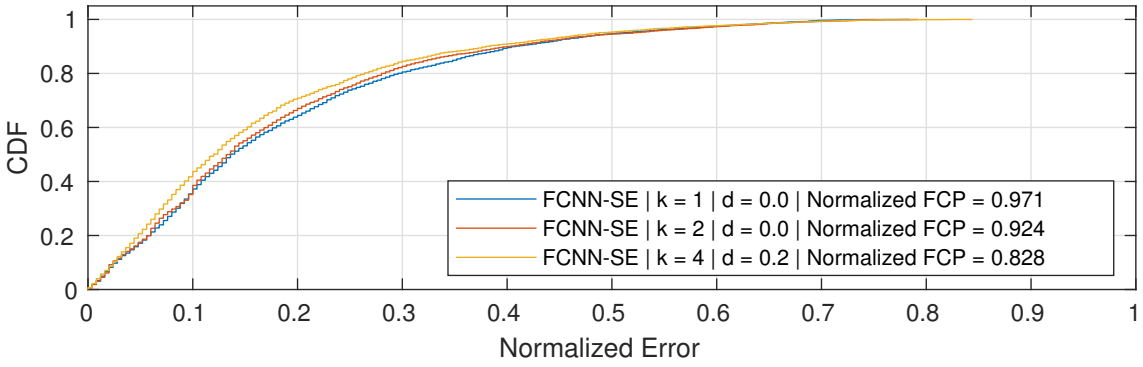


Figure 4.9: CDF curves for the best performing FCNN-SE network for each level of complexity.

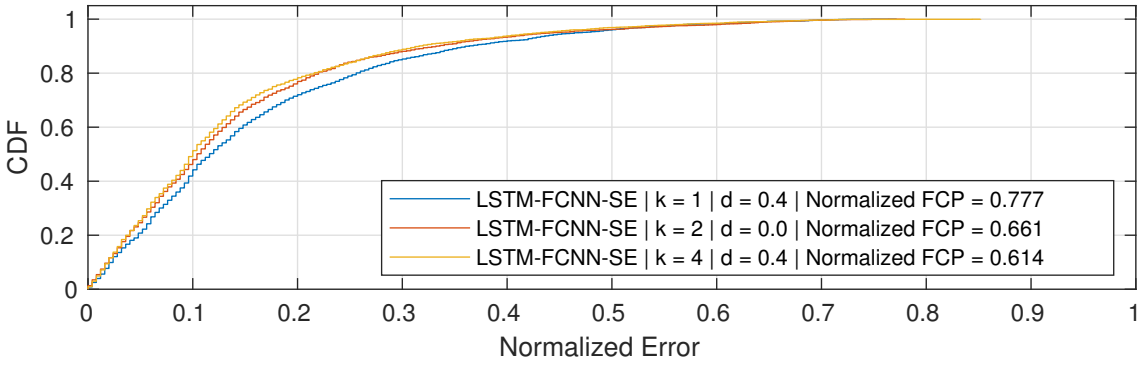


Figure 4.10: CDF curves for the best performing LSTM-FCNN-SE network for each level of complexity.

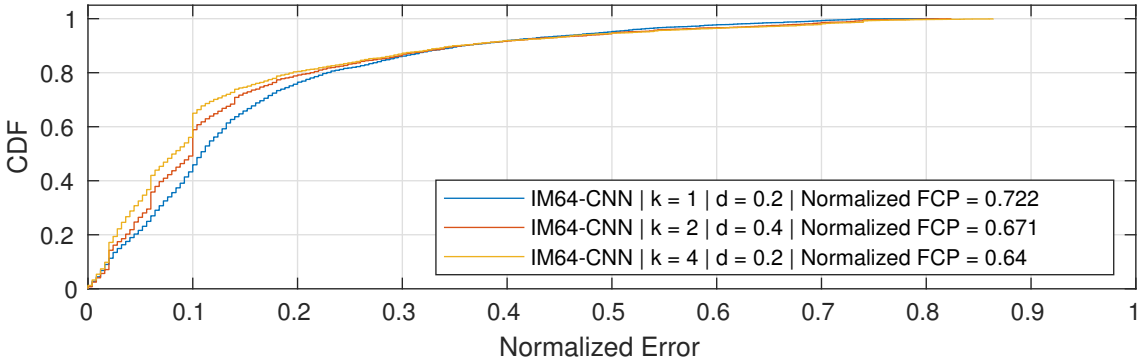


Figure 4.11: CDF curves for the best performing IM64-CNN network for each level of complexity.

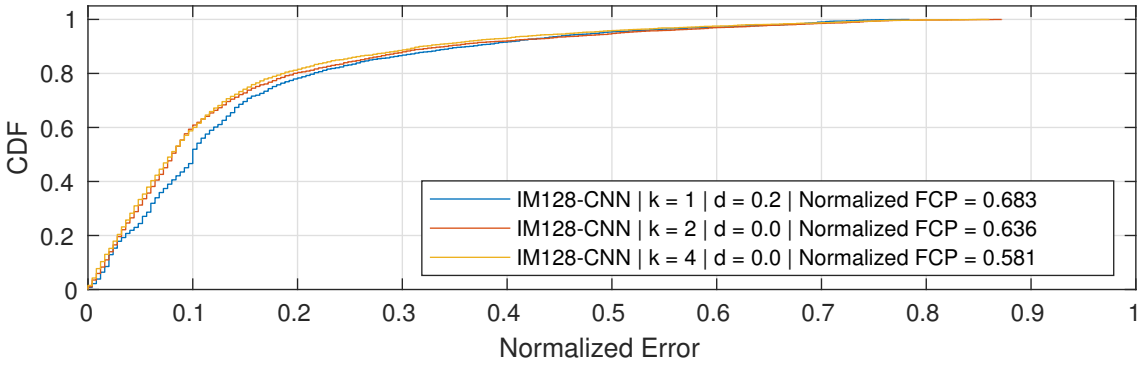


Figure 4.12: CDF curves for the best performing IM128-CNN network for each level of complexity.

Looking at the CDF plots it is clear that the largest difference between low and high complexity networks is in the small error range. For all networks, the percentage of errors larger than 0.5 in the normalized scale is very similar for different complexities. For most of the architecture the major difference between the curves is in the lower half of the error range. Unfortunately, this range of errors is also the area where improvements has the smallest impact on the FCP performance of the network.

By examining where the CDF curves terminate on the right hand side, it can be determined what the maximum prediction error was for the validation data set. Using this approach in figures 4.7 through 4.12 above yields that high complexity networks often have larger maximum prediction errors even though the overall FCP performance is better.

4.3 Computational complexity

The number of parameters of all trained neural networks can be seen in table 4.2 below. The FCP for the best performing network of each complexity is also shown.

Table 4.2: Neural network size and predictive performance of the best performing neural network of each architecture and complexity.

	NOP	Normalized FCP
FCNN		
$k = 1$, $d = 0.0$	1473	1.0051
$k = 2$, $d = 0.0$	4969	0.9693
$k = 4$, $d = 0.2$	18105	0.9426
LSTM-FCNN		
$k = 1$, $d = 0.2$	10953	0.7705
$k = 2$, $d = 0.4$	24441	0.6814
$k = 4$, $d = 0.4$	59097	0.6346
FCNN-SE		
$k = 1$, $d = 0.0$	1633	0.9706
$k = 2$, $d = 0.0$	5609	0.9328
$k = 4$, $d = 0.2$	20665	0.8283
LSTM-FCNN-SE		
$k = 1$, $d = 0.4$	11113	0.7769
$k = 2$, $d = 0.0$	25081	0.6608
$k = 4$, $d = 0.4$	61657	0.6141
IM64-CNN		
$k = 1$, $d = 0.2$	10513	0.7223
$k = 2$, $d = 0.4$	38025	0.6711
$k = 4$, $d = 0.2$	144121	0.6404
IM128-CNN		
$k = 1$, $d = 0.2$	22801	0.6833
$k = 2$, $d = 0.0$	87177	0.6364
$k = 4$, $d = 0.0$	340729	0.5812

In order to study the relation between the network complexity and performance, the two metrics can be visualized together. In figure 4.13 below, the normalized FCP of each network is plotted against the NOP. For clarity, only the best performing network of each complexity is shown.

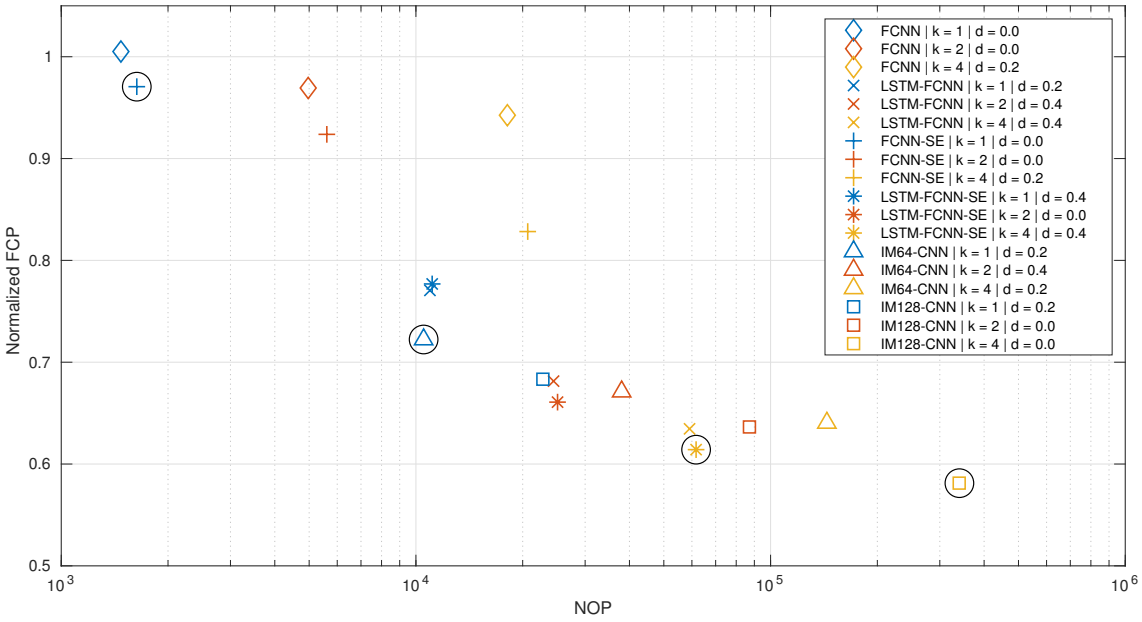


Figure 4.13: Normalized FCP of the trained neural networks plotted against the NOP. Networks chosen for further analysis are circled.

As can be seen, some networks have a performance advantage against others without suffering from any notable increase in complexity. For example, FCNN-SE consistently perform better than FCNN while increasing the NOP very little. Similarily, LSTM-FCNN-SE networks achieve equal or lower FCP values than LSTM-FCNN with an insignificant increase in complexity. When it comes to the image based networks it can be seen that the FCP is trending downwards when the NOP increases. However, it is also seen that the highest complexity IM64-CNN network actually performs worse and has a higher NOP than the middle complexity IM128-CNN network. This is another indication that the increase in image resolution has a positive effect on the predictive performance.

Based on this plot, a small selection of networks is made to facilitate retraining and deeper analysis. The networks are chosen based on their predictive performance, but also to get a wide spread of complexities. As such, the networks used for analysis of varying batch size and training set size are

- FCNN-SE, $k = 1$, $d = 0.0$
- LSTM-FCNN-SE, $k = 4$, $d = 0.4$
- IM64-CNN, $k = 1$, $d = 0.2$
- IM128-CNN, $k = 4$, $d = 0.0$

as can be seen in figure 4.13 where the markers of the selected networks are circled.

4.4 Varying batch size

The selected networks are then retrained on data sets generated with different batch sizes using the same training settings as before. The resulting CDF plots for each network type and batch sizes of 1, 3 and 5 can be seen in figures 4.14 through 4.17.

The CDF curves with a batch size of 1 are the same as previously, but are included in the plots for easy comparison.

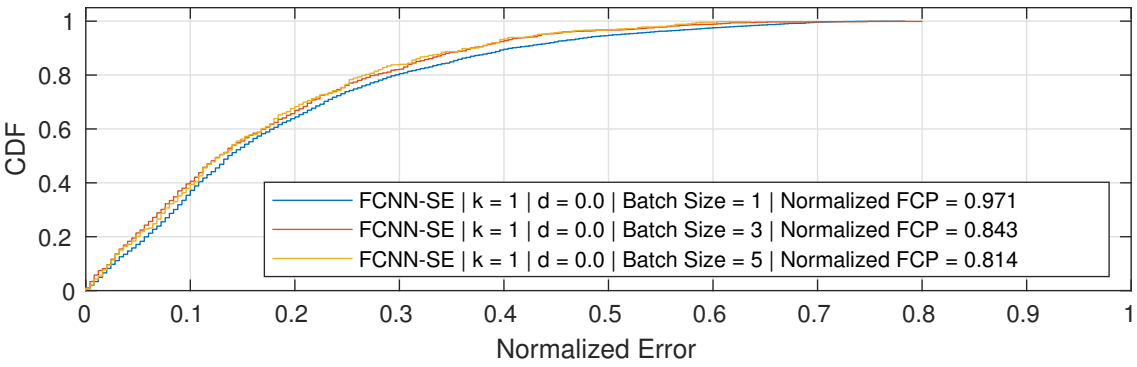


Figure 4.14: CDF curves with varying sweep batch size for the FCNN-SE network with $k = 1$ and $d = 0.0$.

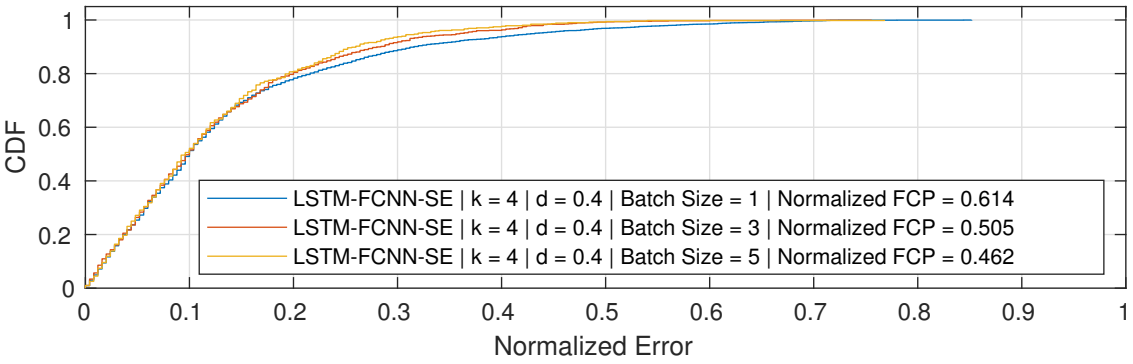


Figure 4.15: CDF curves with varying sweep batch size for the LSTM-FCNN-SE network with $k = 4$ and $d = 0.4$.

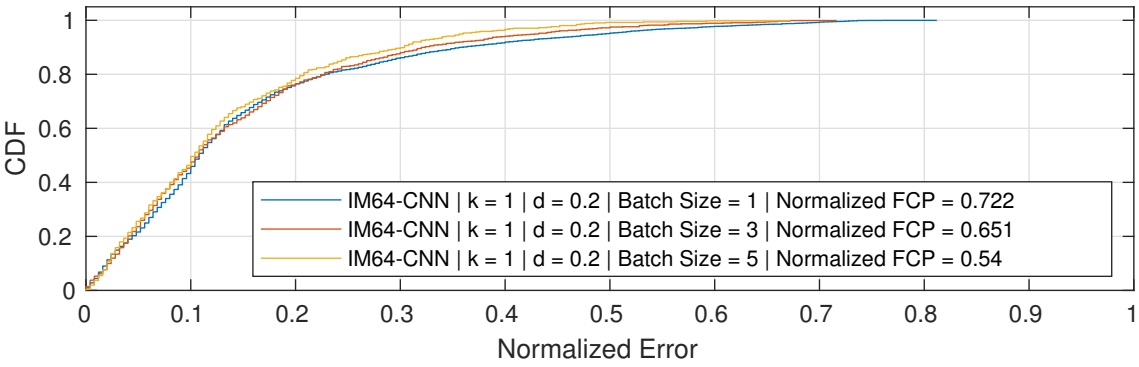


Figure 4.16: CDF curves with varying sweep batch size for the IM64-CNN network with $k = 1$ and $d = 0.2$.

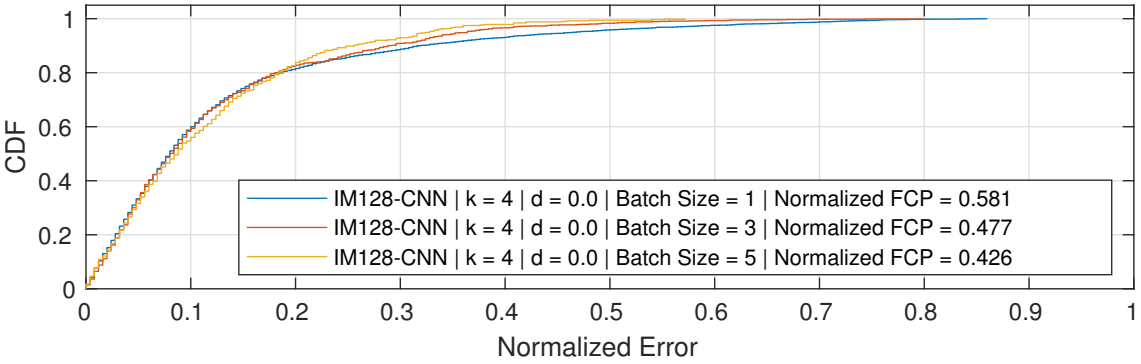


Figure 4.17: CDF curves with varying sweep batch size for the IM128-CNN network with $k = 4$ and $d = 0.0$.

For all the networks evaluated here, the FCP is significantly lower with larger batch sizes. This is inline with what has been observed for traditional machine learning approaches applied on this data set. It can be seen that when using batches of 3 sweeps, the FCP is reduced by 10% to 18% depending on the network. Increasing the batch further, up to 5, improves the performance even more. The further reduction of FCP, compared to a batch size of 3, then ranges from 4% to 18%. It can also be seen that for increasing batch size, the performance gains are mostly in the large error range. Moreover, for all the networks except FCNN-SE the maximum prediction error on the test set is reduced notably.

4.5 Learning curves

The results of retraining the networks with varying size of the training set can be seen in figures 4.18 through 4.21 below.

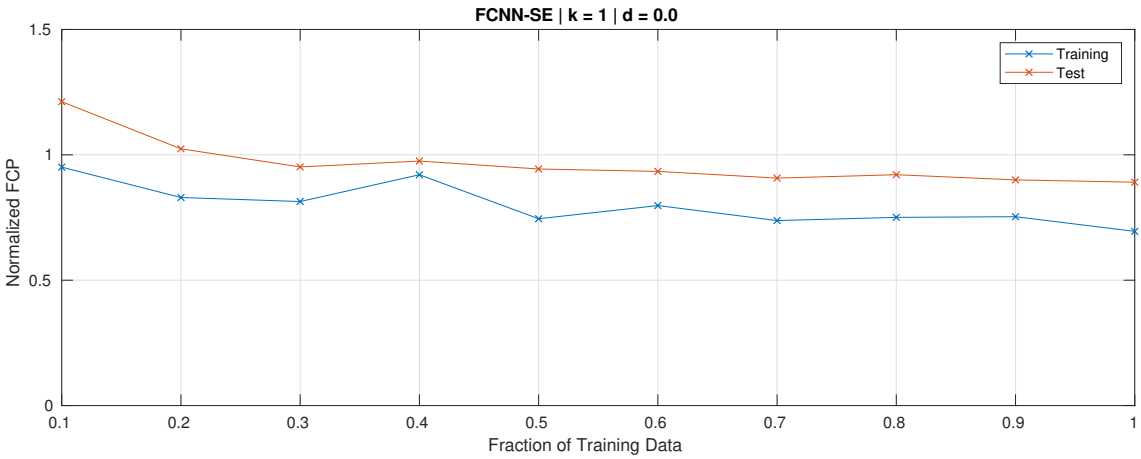


Figure 4.18: Learning curve trajectory for the FCNN-SE network with $k = 1$ and $d = 0.0$.

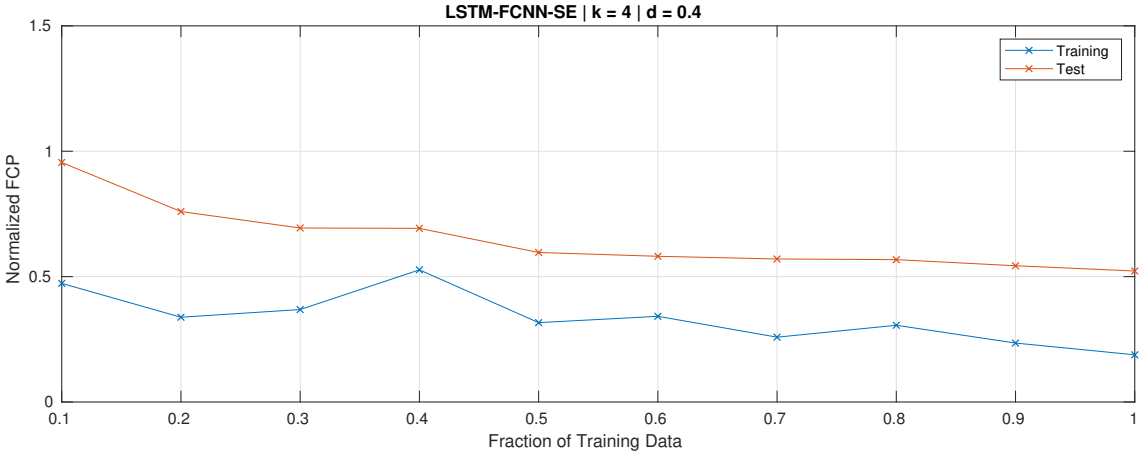


Figure 4.19: Learning curve trajectory for the LSTM-FCNN-SE network with $k = 4$ and $d = 0.4$.

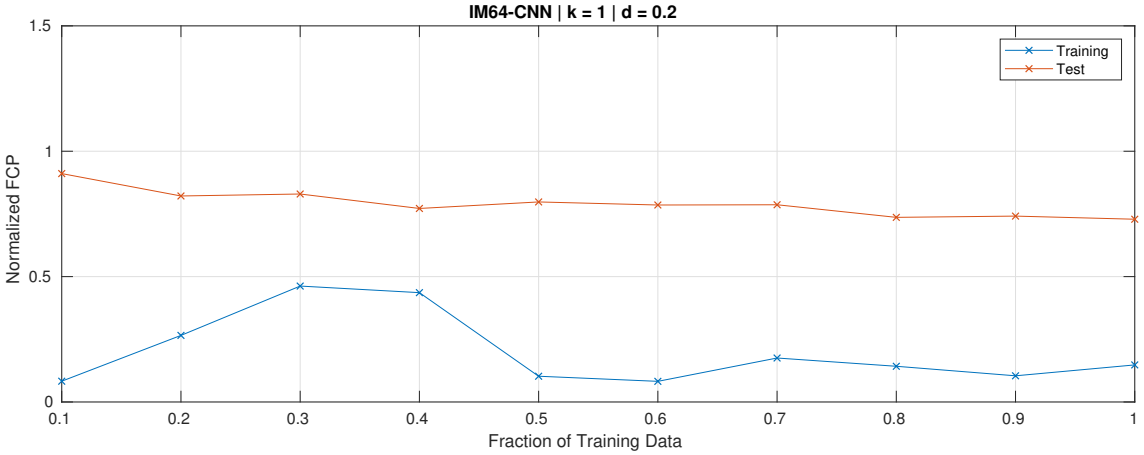


Figure 4.20: Learning curve trajectory for the IM64-CNN network with $k = 1$ and $d = 0.2$.

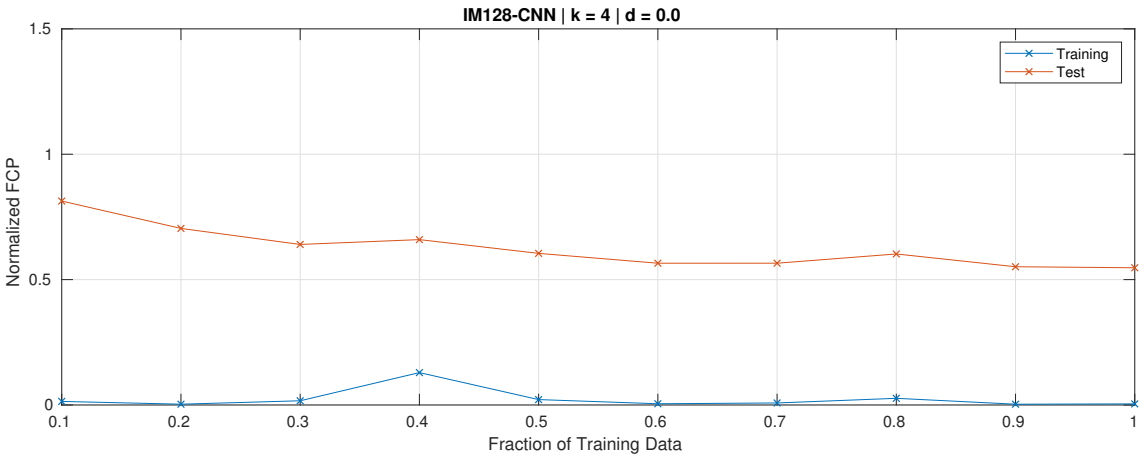


Figure 4.21: Learning curve trajectory for the IM128-CNN network with $k = 4$ and $d = 0.0$.

In all of the above learning curves the test performance is trending downwards when a larger training set is used. There is no clear indication that the performance plateaus when the number of training observations increase. Instead, the learning curves above are an indication that introducing more training data could potentially increase the predictive performance of the networks further.

4.6 Varying data splits

Retraining the small selection of networks on three new splits of the data results in the normalized FCP values seen in table 4.3 below.

Table 4.3: Normalized FCP for a selection of networks trained and evaluated on four different permutations of the same data set.

	I	**II**	**III**	**IV**
FCNN-SE $k = 1$, $d = 0.0$	0.9706	0.9588	0.9986	0.9679
LSTM-FCNN-SE $k = 4$, $d = 0.4$	0.6141	0.5970	0.6374	0.5972
IM64-CNN $k = 1$, $d = 0.2$	0.7223	0.7122	0.7576	0.7138
IM128-CNN $k = 4$, $d = 0.0$	0.5812	0.5851	0.5853	0.5876

As can be seen in the table, retraining the networks on different splits of data changes the performance outcome very little. The most significant difference can be seen in the third split, where the normalized FCP is slightly higher than for other splits. It can also be seen that the highest complexity network, IM128-CNN, is affected very little by changing data split. Furthermore, the effects of the data splits seem to be consistent for all networks trained here. The performance is affected in the same manner for all networks when the data permutation changes, with the exception of IM128-CNN which is largely unaffected.

5
Discussion

The goal of this project was to evaluate the feasibility of using neural networks to predict the optimal roof deflector position on trucks. Results indicate that neural networks able to successfully predict the optimal position and achieve normalized FCP values ranging from 1 to 0.58 depending on the complexity of the network. Note that a normalized FCP of 1 corresponds to the same performance as the baseline algorithm and any lower value is an improvement. There is a strong trend in the results that more complex networks yields better predictive performance. This is a quite typical result when applying machine learning algorithms, as more degrees of freedom tends to yield closer fit to the training data. However, there is often a cutoff where the flexibility of the model is so high that noise in the training data starts being captured by model. This can be clearly seen in the results where the high complexity models almost perfectly fit the training data, i.e. training FCP is very close to zero.

In order to combat the tendencies of overfitting, dropout layers were used. Results show that the fit against training is consistently reduced with increasing dropout rate for all network types. However, whether the reduction of overfitting actually increases performance depends on the architecture of the network. FCNN and FCNN-SE networks showed little overfitting for all complexities and dropout had very little impact on the resulting validation performance. The LSTM-FCNN and LSTM-FCNN-SE networks had much more overfitting. However, this was successfully reduced with dropout, which also increased the validation performance of the networks augmented with an LSTM block. Tendencies of overfitting was the most clear in the image base networks. Regardless of complexity and image resolution, the networks reached almost zero training loss when no dropout was used. For the lower image resolution of 64, performance was improved when dropout was introduced to the training procedure. With networks trained on images of size 128 the results were quite different as there was no clear performance gain by reducing overfitting through dropout layers.

There are strong indications from the analysis of the time series based networks that augmenting the baseline FCNN architecture with SE and LSTM blocks improves the quality of the predictions significantly. Out of the two, the LSTM block yields the highest performance gains. Unfortunately, it also has a very large impact on the NOP, which is increased by 300% to 700% depending on the network complexity. This increase in NOP does however push the normalized FCP to 0.6346 from 0.9426 when augmenting the best performing FCNN network with an LSTM block. The performance gains of the SE augmentation is not as substantial, but the reduction of normalized FCP on the same original FCNN network is 0.114. Though,

the cost in NOP is significantly less and it only increases by around 10% to 15% depending on the network complexity.

Another interesting results is that the base complexity LSTM-FCNN network significantly outperforms the highest complexity FCNN and FCNN-SE networks in terms of both FCP and NOP. In other words, unless an exceptionally low complexity model is desired, it is always preferable to implement the LSTM augmentation for this type of networks.

The image based networks have shown very good results in terms of FCP. The networks trained on images with higher resolution achieve lower FCP across the board, but also use a higher NOP to achieve those numbers. More specifically, a higher image resolution results in a reduction of the normalized FCP of 0.035 to 0.06 depending on the complexity chosen. However, this performance comes at the cost of increasing the NOP by approximately 200%. Nonetheless, the behaviour of both networks types are very similar as the FCP decreases steadily with increasing NOP.

When comparing networks compared on time series data and image data, there is no clear winner. For example, the IM64-CNN architecture outperforms both LSTM-FCNN and LSTM-FCNN-SE in both FCP and NOP when the base complexity is used. For $k = 2$ and $k = 4$, the results are quite different as both LSTM-FCNN and LSTM-FCNN-SE reached lower FCP than any image based network with a comparable NOP. The lowest FCP overall is reached by an IM128-CNN networks. It does however have around a 550% higher NOP and only yields 0.033 lower normalized FCP than the closest performing network, which is of type LSTM-FCNN-SE.

Introducing batches of sweeps during testing has been shown to increase the predictive performance of the networks. Going from predicting on single sweeps to predicting on batches of three sweeps reduces the normalized FCP substantially. For FCNN-SE the normalized FCP is reduced by as much as 0.13, while for the other network the reduction ranges from 0.07 to 0.11. Using a total of five sweeps in each batch further reduces the FCP of the neural networks. Compared to using batches of three, the normalized FCP is then reduced by 0.03 to 0.11 depending on the chosen network. This result is somewhat expected as this trend in improving performance has been observed for the baseline algorithm. By using more sweeps for each prediction, a single bad observation does not necessarily lead to a bad prediction. The other sweeps in the batch may compensate for the bad prediction as the individual predictions are averaged. Likely as a result of this, the maximum prediction error on the entire test set was smaller when batches of sweeps were used. Furthermore, it was observed that the performance mostly improved in the large error range, which is why the FCP is significantly better with higher batch sizes.

The effects of varying the size of the training data set was investigated by producing learning curves for the neural networks. Results show that there is a clear trend of improved predictive performance when the number of observations in the training set grew. The trend indicates that gathering more data to be used for training neural networks is a potential way to improve the performance further. This type of dependence on the training set size is quite typical for neural networks in general. As networks have a very large number of degrees of freedom and are highly flexi-

ble, more data can be a way to reduce overfitting and improve generalization of the models. A more inflexible model may have yielded a learning curve that plateaus, indicating that there is not that much to gain from introducing more data. However, the learning curves for the neural networks gave no such indication which reinforces that adding more observations to the data set should yield better predictions.

It was also investigated if the permutation of data used throughout this work was particularly biased towards better or worse performance. By retraining a selection of networks on new random splits of the data, a small spread in the achieved performance could be observed. The difference between various data splits was very small in comparison to the absolute normalized FCP values. The small number of permutations of data used in this work is not enough to give statistical credibility to the achieved performance results. However, the fact that performance was very similar across the board is an indication that the achieved performance numbers are legitimate and not the result of a particularly favorable data permutation.

6
Conclusion

This thesis set out to investigate the potential of using neural networks to predict the optimal roof deflector position on trucks. To accomplish this, several deep learning methodologies were researched, implemented and evaluated.

A literature survey identified that most time series classification algorithms use either the raw time series or utilize various image transformations. As such, both methodologies were implemented in this work. The time series based approach is a complete end-to-end algorithm that requires no data pre processing or feature engineering. The image based approach uses two image transformations procedures, Gramian angular fields and recurrence plots, to derive the input to the neural network predictor. Both approaches were shown to yield good predictive performance, with the image based networks reaching slightly better scores. However, these scores were achieved using a higher number of model parameters and the image based approach also adds another layer of complexity in terms of pre processing.

Despite the exploratory nature of this thesis, results have shown that deep convolutional neural networks can be used to successfully predict the optimal position of roof deflectors. The neural network based approach yield a lower average fuel consumption increase than the traditional machine learning approach that it was compared against. Moreover, a comparison of the number of parameters of all trained neural networks was carried out. It was shown that if computational complexity is a limiting factor, the number of network parameters can be greatly reduced at only a small cost of predictive performance. This makes the proposed prediction algorithm feasible for deployment on resource constrained hardware.

6.1 Future work

There are several topics in this thesis that would be suitable for further analysis in future work. While the chosen neural networks types evaluated in this work are inline with current literature, there are endless more possibilities to explore in regards to deep machine learning. More types of networks, across a broader range of complexities, could be analyzed to find better performing architectures.

The data set used in this project was somewhat limited, both in terms of size and scope. It has been shown numerous times before that deep neural networks can be made to achieve better performance simply by increasing size of the training data set. Another way to approach this is to add more signals to the input data than just the actuators current consumption. It would be interesting to see if instantaneous fuel consumption or other drivetrain signals could be used to identify changes in the

trucks air resistance to find the optimal roof deflector position. This approach is of course limited by the quality and availability of such data, as it would be feasible to train the networks with signal data that is not available to the algorithm at prediction time.

Another topic that could be evaluated further is the feasibility of deploying the neural network based algorithms on an embedded device with resource constrained hardware. By going into detail on the preparation and deployment of a neural network, factors such as computation time and memory requirements of calculating predictions on-the-fly could be evaluated. Furthermore, it could be evaluated how performance is affected by quantization of network parameters, as this may be a prerequisite for being able to fit the trained networks in an embedded device.

Bibliography

[1] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh, "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances," *Data Mining and Knowledge Discovery*, vol. 31, pp. 606–660, feb 2017.

[2] S. Padmavathi and E. Ramanujam, "Naïve Bayes Classifier for ECG abnormalities using Multivariate Maximal Time Series Motif," *Procedia Computer Science*, vol. 47, pp. 222–228, jan 2014.

[3] B. Chakraborty, "Feature selection and classification techniques for multivariate time series," in *Second International Conference on Innovative Computing, Information and Control, ICICIC 2007*, pp. 42–42, IEEE, sep 2008.

[4] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: a review," sep 2018.

[5] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 2017-May, pp. 1578–1585, nov 2017.

[6] F. Karim, S. Majumdar, H. Darabi, and S. Chen, "LSTM Fully Convolutional Networks for Time Series Classification," *IEEE Access*, vol. 6, pp. 1662–1669, sep 2017.

[7] Z. Cui, W. Chen, and Y. Chen, "Multi-Scale Convolutional Neural Networks for Time Series Classification," *Journal of Inorganic Materials*, vol. 24, pp. 909–914, nov 2016.

[8] Z. Wang and T. Oates, "Encoding Time Series as Images for Visual Inspection and Classification Using Tiled Convolutional Neural Networks," 2015.

[9] Z. Wang and T. Oates, "Imaging time-series to improve classification and imputation," in *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2015-Janua, pp. 3939–3945, may 2015.

[10] Y. Cho, R. A. Ohm, I. V. Grigoriev, and A. Srivastava, "Fungal-specific transcription factor AbPf2 activates pathogenicity in Alternaria brassicicola," *Plant Journal*, vol. 75, pp. 498–514, sep 2013.

[11] N. Hatami, Y. Gavet, and J. Debayle, "Classification of Time-Series Images Using Deep Convolutional Neural Networks," oct 2017.

[12] R. K. Tripathy and U. Rajendra Acharya, "Use of features from RR-time series and EEG signals for automated classification of sleep stages in deep neural network framework," *Biocybernetics and Biomedical Engineering*, vol. 38, pp. 890–902, jan 2018.

[13] R. A. Cohen, "Attention," *Encyclopedia of the Neurological Sciences*, pp. 303–313, sep 2014.

[14] D. Jaiswal, C. B. Prasannan, J. I. Hendry, and P. P. Wangikar, "SWATH Tandem Mass Spectrometry Workflow for Quantification of Mass Isotopologue Distribution of Intracellular Metabolites and Fragments Labeled with Isotopic 13 C Carbon," *Analytical Chemistry*, vol. 90, pp. 6486–6493, sep 2018.
[15] M. Hüsken and P. Stagge, "Recurrent neural networks for time series classification," *Neurocomputing*, vol. 50, pp. 223–235, jan 2003.
[16] P. Malhotra, V. TV, L. Vig, P. Agarwal, and G. Shroff, "TimeNet: Pre-trained deep recurrent neural network for time series classification," jun 2017.
[17] G. Gu and G. Gu, *System Identification*. Prentice Hall, 2012.
[18] E. R. Ziegel, *The Elements of Statistical Learning*, vol. 45 of *Springer Series in Statistics*. New York, NY, USA: Springer New York Inc., 2003.
[19] A. Ng, *Machine Learning Yearning*. Online Draft, 2017.
[20] G. Lewicki and G. Marino, "Approximation by superpositions of a sigmoidal function," *Zeitschrift fur Analysis und ihre Anwendung*, vol. 22, pp. 463–470, dec 2003.
[21] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, pp. 251–257, jan 1991.
[22] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The Expressive Power of Neural Networks: A View from the Width," sep 2017.
[23] B. Hanin, "Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations," aug 2017.
[24] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," nov 2012.
[25] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," sep 2014.
[26] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the Limits of Language Modeling," feb 2016.
[27] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," nov 2015.
[28] Y. A. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, "Efficient backprop," 2012.
[29] S. Wiesler, R. Schluter, and H. Ney, "A convergence analysis of log-linear training and its application to speech recognition," in *2011 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2011, Proceedings*, pp. 1–6, IEEE, dec 2011.
[30] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How Does Batch Normalization Help Optimization?," may 2018.
[31] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," feb 2015.
[32] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," jul 2012.
[33] B. Mele and G. Altarelli, "Lepton spectra as a measure of b quark polarization at LEP," *Physics Letters B*, vol. 299, no. 3-4, pp. 345–350, 1993.
[34] J. Hu, L. Shen, and G. Sun, "Squeeze-and-Excitation Networks," sep 2018.

[35] E. Garcia-Ceja, M. Z. Uddin, and J. Torresen, "Classification of Recurrence Plots' Distance Matrices with a Convolutional Neural Network for Activity Recognition," in *Procedia Computer Science*, vol. 130, pp. 157–163, Elsevier, jan 2018.
[36] J. P. Eckmann, O. Oliffson Kamphorst, and D. Ruelle, "Recurrence plots of dynamical systems," *Epl*, vol. 4, pp. 973–977, nov 1987.
[37] Python, "Python 3.6.0."
[38] Tensorflow, "Tensorflow API Documentation," 2018.
[39] Keras, "Keras Documentation," 2017.
[40] F. Karim, S. Majumdar, H. Darabi, and S. Harford, "Multivariate LSTM-FCNs for Time Series Classification," jan 2018.
[41] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," dec 2014.
[42] Q. Meng, W. Chen, Y. Wang, Z. M. Ma, and T. Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," sep 2019.